



*GraphQL Security*  
*OWASP YVR 2020*



*Hello!*

**I'm Don Burks**

Technical Lead @ **Sphere**

You can find me at @don\_burks



# *Some Assumptions*

You understand what GraphQL is...  
and isn't.

There is an implementation of GraphQL in  
your present or near future

AppSec is something you know is important.

“

*Just because  
it is new,  
that does not  
mean that  
it is secure.*

“

*Ask the MongoDB  
community.*

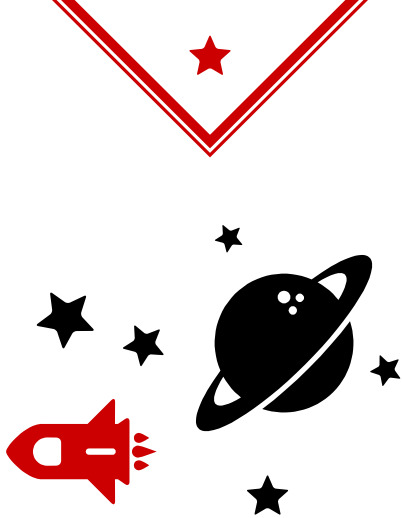


mongoDB



*Tips for securing your GraphQL*

- ◆ Route change
- ◆ Introspection
- ◆ Authentication
- ◆ Depth / Complexity
- ◆ Schema generation




# *Route Change*

Many things we do as developers are  
conventions, not requirements



## **/graphql**

This is the default. It is a convention that has been adopted as the go-to endpoint for all GraphQL implementations.

This makes it a target. 

## **/fluffybunny**

Not a standardly enumerated route. Works just as well as the default. Neither the client, nor the server, is going to care what route the request comes in on, as long as it is a well-formed request.





*Trust  
the bunny*

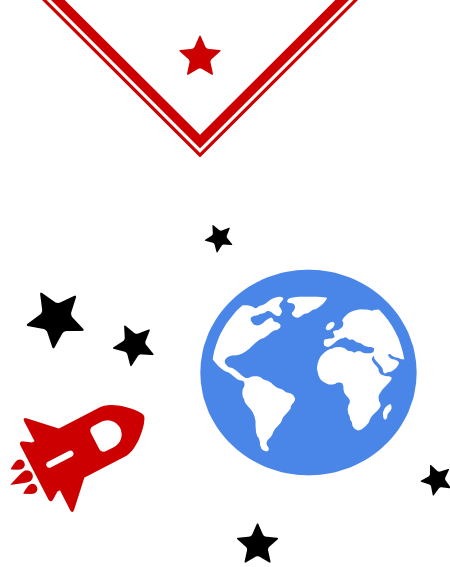


*ALSO!*

**Disable /graphql**

**Yes**, in *all* env's.

Tools such as graphql-ide or Insomnia  
are better.



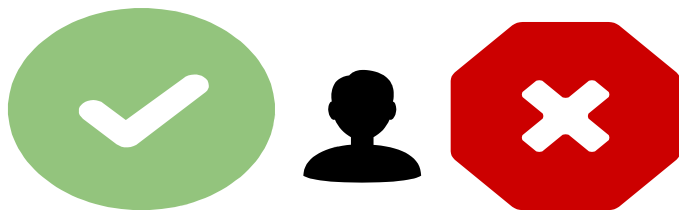
# *Introspection*

Great when you're alone. Not so great when  
you're standing in front of 7 billion people.



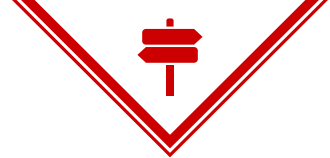
Disable introspection in your testing and production environments.

- ◆ Apollo and fastify-gql now do this by default (in prod)
- ◆ Test for introspection leakiness in your testing env



# *Authentication*

This tends to be a big mistake I see in new GraphQL implementations.



## *Layers of Authentication*

### **JWT**

JSON Web Tokens passed in the Authorization header can be checked at the *context* level with each query.

Just like an API.

### **ACL**

Access Control means that admin queries are restricted to admin accounts. It means resource ownership and / or edit privileges are checked.

### **Edges**

Don't forget to add auth and / or ACL to the resolvers that facilitate your edges. A malicious attacker could easily exploit this to access leaky data.

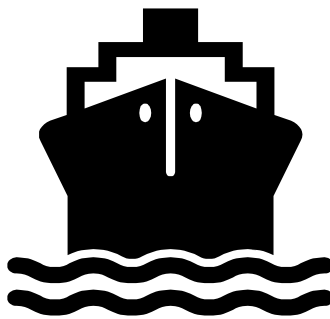
```
type User {  
  
    id: ID  
  
    email: String  
  
    username: String  
  
    admin: Boolean  
  
    createdAt: String  
  
    updatedAt: String  
  
    lastLogin: String  
  
}
```

```
type Post {  
  
    id: ID  
  
    title: String  
  
    body: String  
  
    author: User  
  
    createdAt: String  
  
    updatedAt: String  
  
}
```

```
Post: {  
  author: (post) => {  
    return someDB.select("*").  
      .from("users")  
      .where("id", post.author_id)  
      .limit(1);  
  })  
}
```



```
Post: {
  author: (post, args, context) => {
    if (context.user.admin || context.user.id
=== post.author_id) {
      return someDB.select("*").
        .from("users")
        .where("id", post.author_id)
        .limit(1);
    }
  })
}
```



# *Depth / Complexity*

Easier than you think.

More important than you realize.



## *Different types of complicated queries*

### **Depth**

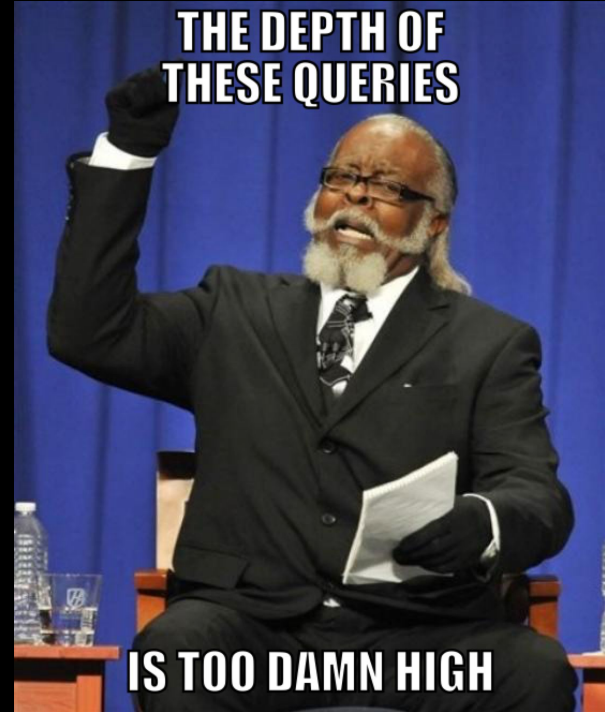
Is the number of edges your query is trying to access.

Too much depth can DDOS your server due to overloading your data store.

### **Complexity**

Some queries may have extreme complexity to them, and should be evaluated accordingly. This involves queries doing heavy joins, aggregations, or retrieving data from external APIs.

```
query {
  users {
    posts {
      user {
        posts {
          user {
            posts {
              user {
                posts {
                  user {
                    posts {
                      id
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```



```
query {  
  users(first: 50) {  
    posts(last: 10) {  
      id  
      title  
      body  
    }  
  }  
}
```

**50 Nodes**

**+ 50 \* 10 Nodes**

**= 550 Nodes**

```
query {  
  users(first: 5000) {  
    posts(last: 100) {  
      id  
      title  
      body  
    }  
  }  
}
```

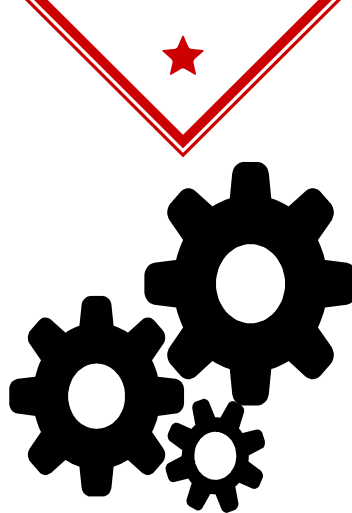
**5000 Nodes**  
**+ 5000 \* 100 Nodes**

**= 505,000 Nodes!!!**



5


If your query is deeper than this, I'm not sure that query depth is your biggest issue.



# *Schema Generation*

Hey, this is so cool!  
It hacked my site for me!





*If it seems  
magical,  
It is  
probably  
dangerous*



## *Generators*

One of the more dangerous approaches to implementing GraphQL by using a tool to auto-generate the SDL.

- ◆ Translates **all** SQL table fields into SDL schema fields
- ◆ Auto-creates queries and mutations to accomplish CRUD functions



**REST with SPRINKLES!!**

[https://img.webmd.com/dtmcms/live/webmd/consumer\\_assets/site\\_images/article\\_thumbnails/recipes/chocolate\\_pudding\\_sprinkle\\_cones\\_recipe/650x350\\_chocolate\\_pudding\\_sprinkle\\_cones\\_recipe.jpg](https://img.webmd.com/dtmcms/live/webmd/consumer_assets/site_images/article_thumbnails/recipes/chocolate_pudding_sprinkle_cones_recipe/650x350_chocolate_pudding_sprinkle_cones_recipe.jpg)



*Design your SDL Schema, don't generate it!*

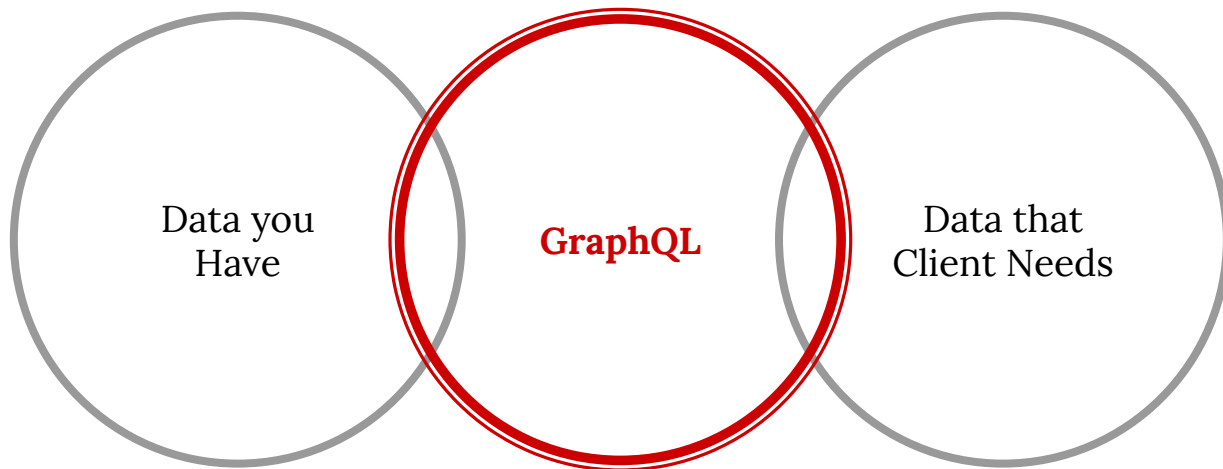
**SDL**

**Queries**

**Mutations**



*You get the opportunity to CRAFT your schema*





## *Summary*

- ◆ Send your *authenticated* query...
- ◆ To a back-end with a *thoughtful* schema...
- ◆ Where the *depth* and *complexity* are evaluated...
- ◆ And the *endpoint* is non-standard...
- ◆ Before you start thinking that you're secure.



*Thanks!*

**Any questions?**

You can find me at:

@don\_burks · donburks.com

 <https://sphere.guide>