Pete Corey Follow
www.petecorey.com/blog
Jun 13, 2017 · 5 min read

GraphQL NoSQL Injection Through JSON Types

One year ago today, I wrote an article discussing NoSQL Injection and GraphQL. I praised GraphQL for eradicating the entire possibility of NoSQL Injection.

I claimed that because GraphQL forces you to flesh out the entirety of your schema before you ever write a query, it's effectively impossible to succumb to the incomplete argument checking that leads to a NoSQL Injection vulnerability.

> *Put simply, this means that an input object will never have any room for wildcards, or potentially exploitable inputs.* **Partial checking of GraphQL arguments is impossible!**

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

I was wrong.

NoSQL Injection is entirely possible when using GraphQL, and can creep into your application through the use of "partial scalar types".

In this article, we'll walk through how the relatively popular `GraphQLJSON` scalar type can open the door to NoSQL Injection applications using MongoDB.

Custom Scalars

In my previous article, I explained that GraphQL requires that you define your entire application's schema all the way down to its scalar leaves.

These scalars can be grouped and nested within objects, but ultimately every field sent down to the client, or passed in by the user is a field of a known type:

> *Scalars and Enums form the leaves in request and response trees; the intermediate levels are Object types, which define a set of fields, where each field is another type in the system, allowing the definition of arbitrary type hierarchies.*

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Normally, these scalars are simple primitives: `String`, `Int`, `Float`, or `Boolean`. However, sometimes these four primitive types aren't enough to fully flesh out the input and output schema of a complex web application.

Custom scalar types to the rescue!

Your application can define a custom scalar type, along with the set of functionality required to serialize and deserialize that type into and out of a GraphQL request.

A common example of a custom type is the `Date` type, which can serialize Javascript `Date` objects into strings to be returned as part of a GraphQL query, and parse date strings into Javascript `Date` objects when provided as GraphQL inputs.

Searching with JSON Scalars

This is all well and good. Custom scalars obviously are a powerful tool for building out more advanced GraphQL schemas. Unfortunately, this tool can be abused.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Imagine we're building a user search page. In our contrived example, the page lets users search for other users based on a variety of fields: username, full name, email address, etc…

Being able to search over multiple fields creates ambiguity, and ambiguity is hard to work with in GraphQL.

To make our lives easier, let's accept the search criteria as a JSON object using the `GraphQLJSON` custom scalar type:

```
type Query {
    users(search: JSON!): [User]
}
```

Using Apollo and a Meteor-style MongoDB driver, we could write our `users` resolver like this:

```
{
        Query: {
        users: (_root, { search }, _context) => {
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
            return Users.find(search, {
                fields: {
                    username: 1,
                    fullname: 1,
                    email: 1
                }
            });
        }
    }
}
```

Great!

But now we want to paginate the results and allow the user to specify the number of results per page.

We could add `skip` and `limit` fields separately to our `users` query, but that would be too much work. We've already seen how well using the `JSON` type worked, so let's use that again!

```
type Query {
    users(search: JSON!, options: JSON!): [User]
}
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

We've extended our `users` query to accept an `options` JSON object.

```
{
        Query: {
        users: (_root, { search, options }, _context) => {
            return Users.find(search, _.extend({
                fields: {
                    _id: 1,
                    username: 1,
                    fullname: 1,
                    email: 1
                }
            }, options));
        }
    }
}
```

And we've extended our `users` resolver to extend the list of fields we return with the `skip` and `limit` fields passed up from the client.

Exploiting the Search

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Now, for example, our client can make a query to search for users based on their username or their email address:

```
{
    users(search: "{\"username\": {\"$regex\": \"sue\"}, \"email\":
{\"$regex\": \"sue\"}}",
          options: "{\"skip\": 0, \"limit\": 10}") {
        _id
        username
        fullname
        email
    }
}
```

This might return a few users with users with `"sue"` as a part of their username or email address.

But there are problems here.

Imagine a curious or potentially malicious user making the following GraphQL query:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
{
    users(search: "{\"email\": {\"$gte\": \"\"}}",
          options: "{\"skip\": 0, \"limit\": 10}") {
        _id
        username
        fullname
        email
    }
}
```

The entire `search` JSON object is passed directly into the `Users.find` query. This query will return all users in the collection.

Thankfully, a malicious user would only receive our users' usernames, full names, and email addresses. Or would they?

The `options` JSON input could also be maliciously modified:

```
{
    users(search: "{\"email\": {\"$gte\": \"\"}}",
          options: "{\"fields\": {}}") {
        _id
        username
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
            fullname
            email
        }
    }
```

By passing in their own `fields` object, an attacker could overwrite the `fields` specified by the server. This combination of `search` and `options` would return all fields (specified in the GraphQL schema) for all users in the system.

These fields might include sensitive information like their hashed passwords, session tokens, purchase history, etc…

Fixing the Vulnerability

In this case, and in most cases, the solution here is to be explicit about what we expect to receive from the client. Instead of receiving our flexible `search` and `options` objects from the client, we'll instead ask for each field individually:

```
type Query {
    users(fullname: String,
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
            username: String,
            email: String,
            skip: Number!,
            limit: Number!): [User]
}
```

By making the search fields (`fullname` , `username` , and `email`) optional, the querying user can omit and of the fields they don't wish to search on.

Now we can update our resolver to account for this explicitness:

```
{
        Query: {
        users: (_root, args, _context) => {
            let search = _.extend({}, args.fullname ? { fullname } :
{},
                                      args.username ? { username } :
{},
                                      args.email ? { email } : {});
            return Users.find(search, {
                fields: {
                    _id: 1,
                    username: 1,
                    fullname: 1,
                    email: 1
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
                    },
                    skip: args.skip,
                    limit: args.limit
                });
            }
        }
    }
```

If either `fullname`, `username`, or `email` are passed into the query, we'll add them to our query. We can safely dump this user-provided data into our query because we know it's a `String` at this point thanks to GraphQL.

Lastly, we'll set `skip` and `limit` on our MongoDB query to whatever was passed in from the client. We can be confident that our `fields` can't possibly be overridden.

Final Thoughts

Custom scalar types, and the JSON scalar type specifically, aren't all bad. As we discussed, they're a powerful and important tool for building out your GraphQL schema.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

However, when using JSON types, or any other sufficiently expressive custom scalar types, it's important to remember to make assertions about the type and shape of user-provided data. If you're assuming that the data passed in through a JSON field is a string, check that it's a string.

If a more primitive GraphQL type, like a `Number` fulfills the same functionality requirements as a `JSON` type, even at the cost of some verbosity, use the primitive type.

GraphQL Apollo Elixir NoSQL Security

Like what you read? Give Pete Corey a round of applause.
From a quick cheer to a standing ovation, clap to show how much you enjoyed this story.

80 1

Pete Corey
www.petecorey.com/blog

Follow

Responses

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Write a response...

Conversation between Arseny Reutov and Pete Corey.

Arseny Reutov
Jun 19, 2017

I had the same idea while looking through the code of Vulcan.js (http://vulcanjs.org/) which is a framework based on Meteor + GraphQL, but it seems they are immune to JSON injection thanks to lower level mongodb schema validation on behalf of Meteor.

1 1 response

Pete Corey
Jun 20, 2017

Yeah, I've been looking at Vulcan.js as well. Sacha Greif does a great job of locking down his projects and making security a priority.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD