# How to Secure a GraphQL API (The Complete Vulnerability Checklist)

Querying exactly what you want when you want has complex security implications. *This is how to secure a GraphQL API* from abusive queries by clients.

**To properly secure a GraphQL API, servers must timeout, limit depth and limit complexity of queries to mitigate Denial of Service (DoS) attacks. Servers should always whitelist queries whenever applicable. Servers must also throttle all clients and keep track of query costs. API developers should handle authentication with query context to mitigate SQL Injections and Langsec Issues, additionally, developers they must inspect their authorization logic carefully, making sure internal schema stays hidden and the documentation stays up-to-date through iterations.**

We will implement each of the above best-practices in a GraphQL API by following the simple step-by-step checklist below.

1. **Set Query Timeouts**
2. **Limit Query Depth**
3. **Limit Query Complexity**
4. **Whitelist Queries**
5. **Persist Queries**
6. **Throttle all Clients**
7. **Protect GraphQL Endpoint**

Let's dive right in...

## Query Timeouts

This is essentially setting maximum time limits for a each query.

A server configured with a 3 sec timeout would stop execution of any query on immediately the 3 sec window lapses.

While timeouts are easy to implement, the attacker might already gained access by the time the timeout kick in.

Timeouts often used protect against large queries as a last resort.

## Setting Query Timeouts

# Query Depth

Instead of requesting useful data, attackers often submit expensive, nested queries that overload your serve, databases and network denying your web service to other applications.

These complex and expensive queries are usually huge deeply nested.

Unbounded GraphQL queries allow attackers to abuse query depth.

This is a possible DoS vulnerability. With enough depth, this can easily take out your GraphQL server.

Each layer adds overhead to your backend.

## Cyclical Queries

With just an `Author` type that has a list of posts and a `Post` type that allows `Author` retrieval. GraphQL you can rather easily allow complex cyclical queries such as this one:

```
# cyclical query
# depth: 8+
query cyclical {
  author(id: "xyz") {
    posts {
      author {
        posts {
          author {
            posts {
              author {
                ... {
                  ... # more deep nesting!
                }
              }
            }
          }
        }
      }
    }
  }
}
```

## Inline and Named Fragments

In GraphQL, inline and named fragments don't increase depth.

```
# inline fragment
# depth: 1
query inlineShallow {
  authors
  ... on Query {
    posts
  }
}

# named fragment
# depth: 1
query namedShallow {
  ... namedFragment
}

fragment namedFragment on Query {
  posts
}
```

`Maximum Query Depth` is used to limit query depth.

A GraphQL server can then easily reject queries based on depth.

## Limiting Query Depth with `graphql-depth-limit`

At the time of writing this, there is no builtin feature in Apollo GraphQL for handling Query Depth, `graphql-depth-limit` is recommended instead.

Limiting query depth with `express-graphql` and `graphql-depth-limit` is done using `validationRules` as follows:

```
import depthLimit from 'graphql-depth-limit'
import express from 'express'
import graphqlHTTP from 'express-graphql'
import schema from './schema'

const app = express()

const DepthLimitRule = depthLimit(
  4,
  { ignore: [ 'whatever', 'trusted' ] },
  depths => console.log(depths)
)

const graphqlMiddleware = graphqlHTTP({
  schema,
  validationRules: [
    DepthLimitRule,
  ],
})

app.use('/graphql', graphqlHTTP((req, res) => ({
  graphqlMiddleware
})))
```

the first argument in `deepLimit` specifies total depth limit. This will throw back a validation error for queries with a depth of 5 or more.

The second argument is an option object for specifying ignored fields.

A `Maximum Query Depth` of `4` would then consider the query below invalid.

```
# invalid query
# depth: 9+
query cyclical {
  author(id: "xyz") {
    posts {
      author {
        posts {
          author {
            posts {
              author {
                ... {
                  ...
                }
              }
            }
          }
        }
      }
    }
  }
}
```

This is what will be returned to the client:

```
{
  "errors": [
    {
      "message": "'cyclical' exceeds max... depth of 4",
      "locations": [
        {
          "line": ...,
          "column": ...
        }
      ]
    }
  ]
}
```

Since the abstract tree syntax is analyzed statistically, there is no additional load to the GraphQL server, the query does not even execute.

Depth alone cannot cover shallow abusive, that where query complexity comes in.

# Query Complexity

There are fields in our schema that are more expensive to compute that others. Often due to complexity.

Expensive queries should be limited because they add overhead to our GraphQL server and can also be used for DoS attacks.

Query complexity defines and restricts complex field using integers.

Let's denote each field with with an integer signifying complexity, from starting from a default of `1`

```
query simple {
  author(id: "xyz") {      # complexity: 1
    posts(first: 5) {      # complexity: 5
      title               # complexity: 1
    }
  }
}
```

The query would have a complexity of 7. If we had then set a maximum complexity of 5 on our schema, this query would fail.

Note that here, the `posts` more expensive than `author` because complexity is based on arguments.

## Limiting Query Complexity with `graphql-validation-complexity`

Like `graphql-depth-limit` Apollo GraphQL has no builtin feature for limiting query complexity. `graphql-validation-complexity` is recommended instead.

Limiting query complexity with `graphql-validation-complexity` and `express-graphql` is somewhat trivial.

```
import { createComplexityLimitRule } from 'graphql-validation-complexity'
import express from 'express'
import graphqlHTTP from 'express-graphql'
import schema from './schema'

const app = express()

const ComplexityLimitRule = createComplexityLimitRule(1000, {
  scalarCost: 1,
  objectCost: 10, // Default is 0.
  listFactor: 20, // Default is 10.
})

const graphqlMiddleware = graphqlHTTP({
  schema,
  validationRules: [
    ComplexityLimitRule,
  ],
})

app.use('/graphql', graphqlHTTP((req, res) => ({
  graphqlMiddleware
})))
```

The configuration objects are the custom globals for scalars, objects and lists respectively. `objectCost` and `listFactor` have a default of `0` and `10` respectively.

You can also limit query complexity by passing cost factors with `getCost` and `getCostFactor` callbacks in field definitions.

```
const expensiveField = {
  type: ExpensiveItem,
  getCost: () => 60,
};

const expensiveList = {
  type: new GraphQLList(MyItem),
  getCostFactor: () => 100,
};
```

Or in your GraphQL schema definitions as follows:

```
type CustomCostItem {
  expensiveField: ExpensiveItem @cost(value: 50)
  expensiveList: [MyItem] @costFactor(value: 100)
}
```

Query complexity is however, still hard to implement perfectly (particularly with mutations) because complexity is estimated by developers and it often needs to be updated, especially after iterations.

## Calculating Query Costs in GraphQL schema with `graphql-cost-analysis`

`graphql-cost-analysis` makes calculating query costs somewhat easier but allowing custom fine-grained control over specific fields with the `@cost` directive.

Is then parses queries and computes costs.

It allows custom query costs to be dropped right in the GraphQL schema as follows:

```
# you can define a cost directive on a type
type TypeCost @cost(complexity: 3) {
  string: String
  int: Int
}

type Query {
  # will have the default cost value
  defaultCost: Int

  # will have a cost of 2 because this field does not depend on its parent fields
  customCost: Int @cost(useMultipliers: false, complexity: 2)

  # complexity should be between 1 and 10
  badComplexityArgument: Int @cost(complexity: 12)

  # the cost will depend on the `limit` parameter passed to the field
  # then the multiplier will be added to the `parent multipliers` array
  customCostWithResolver(limit: Int): Int
    @cost(multipliers: ["limit"], complexity: 4)

  # for recursive cost
  first(limit: Int): First
    @cost(multipliers: ["limit"], useMultipliers: true, complexity: 2)

  # you can override the cost setting defined directly on a type
  overrideTypeCost: TypeCost @cost(complexity: 2)
  getCostByType: TypeCost

  # You can specify several field parameters in the `multipliers` array
  # then the values of the corresponding parameters will be added together.
  # here, the cost will be `parent multipliers` * (`first` + `last`) * `complexity
  severalMultipliers(first: Int, last: Int): Int
    @cost(multipliers: ["first", "last"])
}

type First {
  # will have the default cost value
  myString: String

  # the cost will depend on the `limit` value passed to the field and the value of `compl
  # and the parent multipliers args: here the `limit` value of the `Query.first` field
  second(limit: Int): String @cost(multipliers: ["limit"], complexity: 2)

  # the cost will be the value of the complexity arg even if you pass a `multipliers` arr
  # because `useMultipliers` is false
  costWithoutMultipliers(limit: Int): Int
    @cost(useMultipliers: false, multipliers: ["limit"])
}
```

# Whitelisting Queries

Query whitelisting has a somewhat limited scope because the list of approved queries needs to be maintained manually.

But, compared to Query Depth and Complexity, it is a more restrictive approach.

It however might not play well, especially with huge public APIs as effectiveness boils down to a list.

The list might needs to be scrutinized thoroughly to make sure it does not whitelist malicious or unwanted queries.

Side effects of whitelisted queries also have to be examined carefully to ensure that legitimate queries or nested fragments are not blocked.

A far better approach is to persist queries.

# Persisting Queries

## Static GraphQL Queries

One of the best practices of GraphQL that has emerged over the years is that it's best to write GraphQL queries as static strings using the GraphQL language.

It better to avoid generating queries dynamically at runtime and it's better to use the GraphQL query language directly than say, through javascript syntax (like tagged template literals) because the best GraphQL tools and integrations rely on the static query language.

One such tool is `persistgraphql` .

Apollo Client uses static queries.

## Persisted Queries

Apollo GraphQL defines persisted queries as a number-to-query-mapping between a server GraphQL Client and a GraphQL Server.

Persisted queries solve the challenges of whitelisting and prevent the client from sending malicious queries while saving significant bandwidth.

Usually, an Apollo Client has to send down every query at runtime to the server as a string, the server will then resolve it and send back some data - an unrestricted query execution of sorts.

Using static queries, queries are determined at build time and saved in a query document before the client is even rendered. This way, a bidirectional mapping between GraphQL documents and generated IDs is established.

After building, the server will only resolve queries that it already knows about.

## Whitelisting and Persisting Queries with `persistgraphql`

`persistgraphql` is a buildtime tool that extracts static GraphQL queries from `.graphql` files.

Each query is then assigned a hash. The hashes are then stored in a JSON object. This is how queries are whitelisted and persisted.

First we install `persistgraphql` CLI tool (together with the Apollo Network Interface).

```
$ npm install --save persistgraphql
```

Then we point the tool to our frontend source directory for `.graphql` files.

```
$ persistgraphql src/
```

Or to a file containing GraphQL

```
$ persistgraphql queries.graphql
```

If it is called on a directory, it will step recursively through each `.graphql` file.

It is also possible to extract GraphQL queries from Javascript using `--extension=js --js` as follows:

```
$ persistgraphql src/index.js --js --extension=js
```

Apollo uses query support by default using it's client network interface.

Only query hashes and are sent to the server, not the query document.

The Apollo Client Network Interface implementation is a replacement of the standard network interface.

If you use the the client network interface, you can roll your own middleware in the GraphQL server as follows:

```
import queryMap from '../extracted_queries.json'
import { invert } from 'lodash'

...

app.use(
  '/graphql',
  (req, resp, next) => {
    if (config.persistedQueries) {
      const invertedMap = invert(queryMap)
      req.body.query = invertedMap[req.body.id]
    }
    next()
  },
)
```

## Automatically Persisting Queries

The above build process of persisting query support with `persistgraphql` gives you an understanding of how automatically persisting queries work - more like peeking under-the-hood.

The alternative is to simply add automatic persisted query link as illustrated below, upgrade to the latest version of Apollo Engine and you're set.

## Automatically Persisting Queries with `apollo-link-persisted-queries`

The `apollo-link-persisted-queries` library is an implementation for use with Apollo Client by using a custom Apollo Link (using `http-link` ).

```
$ npm -install apollo-link-persisted-queries --save
```

Then in your GraphQl server.js

```
import { createPersistedQueryLink } from "apollo-link-persisted-queries"
import { createHttpLink } from "apollo-link-http"
import { InMemoryCache } from "apollo-cache-inmemory"
import ApolloClient from "apollo-client"

// use this with Apollo Client
const link = createPersistedQueryLink().concat(createHttpLink({ uri: "/graphql" }))
const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: link,
})
```

That's it, now your GraphQL server will start whitelisting and persisting queries. You can see the library for configuration options.

# Throttling (Rate Limiting) Clients

Unlike the other steps discussed above, rate limiting usually targets clients making a lot of medium sized queries.

The concept behind rate limiting is fairly simple -- no matter how inexpensive a query is, sending it a significant number of times will inevitably become expensive.

A rate limiter caps how many requests a client can sent in a specific time window.

In most GraphQL APIs, a throttle is used to limit clients from requesting resources too often.

Limiting the amount of requests in GraphQL APIS is not nearly as effective as REST. Because of the nature of GraphQL queries, just a few expensive requests can cause significant overhead

It's not even possible to accurately determine an acceptable number of request because they are determined by the client.

## Rate Limiting Based on Server Time

How long a query takes to execute can be used as an estimate of how expensive it is.

A classic "rate limit algorithm" is the classic token bucket algorithm (or leaky bucket algorithm).

If the maximum server time (bucket size) allowed was set at `1000ms` and clients gain `50ms` of server time / sec (leak rate).

A query that takes `200ms` to complete can only be called 5 times within a second. It would then be blocked on the 6th until more server time is added to the client.

Meanwhile, the client will gain `200ms` in 4 seconds. After which it can call the query only once.

As you might expect, this naturally allocates more server time to less expensive queries that take less server time to compute and vice versa.

Such rate limiting contraints are easy to expressed in GraphQL API docs, but as stated earlier, it is hard to accurately estimate the amount of time a certain query will take (without trying it first) when these are generated by the client.

## Rate Limiting Based on Query Complexity

This usually involves using a rate limiting algorithm (such as the leaky bucket algorithm illustrated above) with query complexity.

Take the following query with a complexity of `7` for example:

```
query simple {
  author(id: "xyz") {      # complexity: 1
    posts {                # complexity: 1
      title                # complexity: 1
    }
  }
}
```

Now if instead of the using maximum server time, we use query cost. We can cap the bucket size at a maximum cost of `9` and say a leak rate of say `0.5` per second.

A client can now only call `simple` 3 times before being blocked, and it needs at least 6 seconds before calling it again.

Github actually uses rate limiting based on complexity on their GraphQL server.

# Protecting the GraphQL Endpoint

Authentication and authorization are widely and adequately covered by resources such as this and this.

## Authentication Best Practices in GraphQL

The authentication best practices in GraphQL from a security standpoint are:

1. Force SSL everywhere
2. Always keep (and frequently inpect) access and error logs.

## Authorization Best Practices in GraphQL

The authorization best practices in GraphQL from a security standpoint are:

1. Checking authorization per node.

2. Factor out data fetching into a separate layer and do the authorization check there.