

Security Assessment

Prepared For

GraphQL Example

Document Revision

Initial Report Prepared By:	AppCheck-NG	Version: 1.0
Reviewed By:	n/a	

Assessment Schedule

Assessment Performed on	Tuesday 09 June 2020
Report Prepared on	Tuesday 09 June 2020

1. CONTENTS

1.	Contents	2
2.	Project Overview	3
3.	Summary of Vulnerabilities: High / Medium	4
	3.1. Graphical Summary.....	5
	3.2. HIGH Impact Vulnerabilities.....	6
	3.3. Medium Impact Vulnerabilities	7
4.	Assessment Results	8
	4.1. Server Side JavaScript injection (error induction check)	8
	4.2. NoSQL Injection (MongoDB Server-Side JS Execution)	10
	4.3. Insecure Direct Object Reference (IDOR)	13
	4.4. Slowloris Denial of Service (DoS)	15
	4.5. Possible Sensitive Data Disclosure (Credentials)	17
	4.6. GraphQL Introspection Successful	18
	4.7. GraphQL Endpoint Detected.....	22
5.	Extended Information.....	24
	5.1. Port Scan	24
6.	Appendix A: Web Application Defensive Strategies	25
	6.1. Validating Input.....	25
	6.2. Authentication	27

2. PROJECT OVERVIEW

The following web applications were defined within the scope for this assessment:

Application URLs / IP Addresses
http://192.168.0.70:4000/graphql?

Vulnerability Impact Ratings

HIGH



HIGH: Successful exploitation could lead to highly privileged access to the target host or cause a denial of service condition.

Vulnerabilities are labelled "HIGH" severity if they have a CVSS base score of 7.0 -10.0.

Medium



Medium: Exploitation of the vulnerability will not directly lead to privileged access to the host, service or data. However, vulnerabilities with a Medium impact can often be combined with other flaws to elevate their impact.

Vulnerabilities will be labelled "Medium" severity if they have a base CVSS score of 4.0-6.9

Low

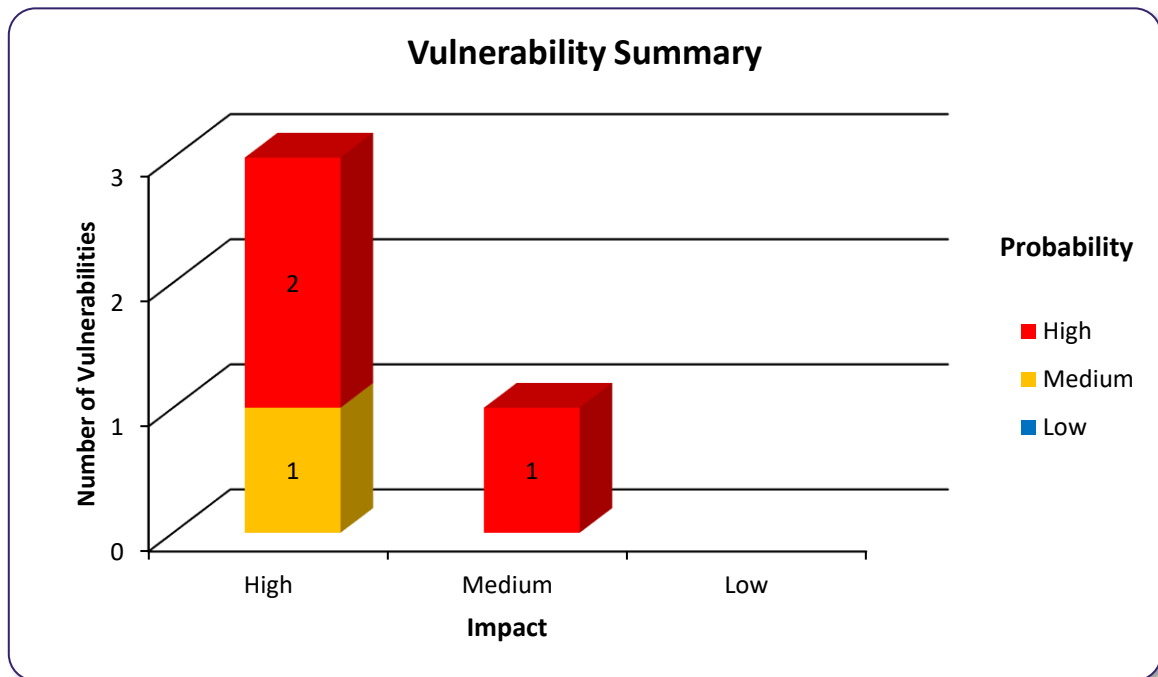


Low: This impact rating is assigned to vulnerabilities that, when exploited in isolation, have a negligible impact on security. Typically vulnerabilities that disclose information that may be useful to the attacker are considered to have a low impact.

Vulnerabilities are labelled "Low" severity if they have a CVSS base score of 0.0-3.9.




3.1. GRAPHICAL SUMMARY

Key findings have been ranked and positioned in the following table according to the relative risk or probability of exploit. Vulnerabilities are split into 3 impact categories: High, Medium and Low. Risk is calculated by comparing the impact vs. the probability of exploit which is represented using colour coding.




3.2. HIGH IMPACT VULNERABILITIES

The following vulnerabilities have been assigned a **HIGH** impact rating. Successful exploitation of these vulnerabilities could lead to highly privileged access to the affected host or data or a denial of service condition.

Impact / Ref	Description	Affected Hosts
 CVSS: 9.4 Impact/Prob: High/High	Server Side JavaScript injection (error induction check) The affected application appears to be vulnerable Server-Side JavaScript injection. This vulnerability is sometimes be referred to as "NoSQL Injection" depending on the root cause of the vulnerability.	http://192.168.0.70:4000
 CVSS: 9.4 Impact/Prob: High/High	NoSQL Injection (MongoDB Server-Side JS Execution) The affected application appears to be vulnerable to NoSQL Injection.	http://192.168.0.70:4000
 CVSS: 6.8 Impact/Prob: High/Medium	Insecure Direct Object Reference (IDOR) Insecure direct object references (IDOR) is a type of access control vulnerability whereby the attacker is able to access restricted data by manipulating a client supplied identifier.	http://192.168.0.70:4000

3.3. MEDIUM IMPACT VULNERABILITIES

The following vulnerabilities have been assigned a **Medium** impact rating.

Impact / Ref	Description	Affected Hosts
 CVSS: 6.8 Impact/Prob: Medium/High	Slowloris Denial of Service (DoS) Slowloris works by opening multiple connections to the targeted web server and keeping them open as long as possible. It does this by continuously sending partial HTTP requests, none of which are ever completed. The attacked servers open more and connections open, waiting for each of the attack requests to be completed.	http://192.168.0.70:4000

4. ASSESSMENT RESULTS

4.1. SERVER SIDE JAVASCRIPT INJECTION (ERROR INDUCTION CHECK)



CVSS Score: 9.4 **CVSS Vector: AV:N/AC:L/Au:N/C:C/I:C/A:N** **Impact/Probability: High/High**

Affected: http://192.168.0.70:4000

The affected application appears to be vulnerable Server-Side JavaScript injection. This vulnerability is sometimes be referred to as "NoSQL Injection" depending on the root cause of the vulnerability.

Server-side code injection vulnerabilities occur when user controllable input is included within a string which is evaluated by a code interpreter. If the user supplied data is not strictly validated, an attacker can exploit this flaw to inject arbitrary code that will be executed by the server.

JavaScript code injection flaws most commonly occur in NoSQL applications that support the use of JavaScript within the database query language. Its also common to find JavaScript injection within applications written in NodeJS where user supplied input is passed to a dangerous function such as eval(), setTimeout(), setInterval() and Function().

Typically this vulnerability is considered high impact. However, in some cases the JavaScript interpreter may be restricted to a small subset of functions which could limit the available attack surface.

For example, when injecting into a MongoDB (above version 2.4) search query, the attacker would typically be restricted to reading data from documents within the queried collection or causing a denial of service condition. Other JavaScript injections, such as those found in NodeJS applications could be exploited to gain full control of the affected server via system command execution.

4.1.1. REMEDIATION

When possible user controllable input should not be included within dynamically evaluated code. Instead, an alternative method should be sought to keep data and code separate. If this isn't possible, data should be strictly validated against a whitelist of known good input.

4.1.2. TECHNICAL OVERVIEW

Vulnerable parameters:

App	URI	Parameter
http://192.168.0.70:4000	/graphql	payload.json.variables.book__id

4.1.3. TECHNICAL ANALYSIS

Example: <http://192.168.0.70:4000/graphql> [param: payload.json.variables.book__id]

Technical Details

The flaw was detected by injecting the following JavaScript payload designed to raise an error containing the token **nbABZunVJG** when successfully executed by the server.

```
'-eval(String.fromCharCode(110,98,65,66,90,117,110,86,74,71))-'
```

Example Exploit

The following exploit payload was used to execute **JSON.stringify(this)** and return the result:

```
'-eval('result'+(function () {var x=JSON.stringify(this),y='';for(i=0;i<x.length;i++){y+=String(x.charCodeAt(i));y+='s'}return y})(); +'endresult')-'
```

Exploit output:

```
{ "__lastres__": {}, "obj": { "_id": { "$oid": "5eb9391c82fcb704cc675843" }, "name": "The Stand", "genre": "Horror", "authorId": "5eb9390982fcb704cc675842", "__v": 0 }, "fullObject": true, "__returnValue": false }
```

4.2. NOSQL INJECTION (MONGODB SERVER-SIDE JS EXECUTION)



CVSS Score: 9.4 CVSS Vector: AV:N/AC:L/Au:N/C:I/C/A:N Impact/Probability: High/High

Affected: http://192.168.0.70:4000

The affected application appears to be vulnerable to NoSQL Injection. NoSQL Injection vulnerabilities occur when client supplied data is included within a NoSQL query in an insecure way. There are several reasons a NoSQL Injection flaw can occur, the most common cases are detailed below.

User Controllable Query

This variant of NoSQL injection occurs when the target application includes JSON properties submitted by the user directly within a NoSQL query. For example, consider a NodeJS & MongoDB application that executes the following query in order to validate a users credentials:

```
User.find({username: username, password: password});
```

The above query will return documents (records) if both the username and password values match a document in the database. If a match is found, the user is deemed to have entered valid credentials and is successfully authenticated.

Due to the compatibility between NoSQL queries and objects used in a variety of programming languages (e.g. JavaScript JSON objects) it can be convenient to read property values from a user supplied object then include them directly within the database query. For example, lets assume that the username and password values in the above query are read from a JSON object submitted via the following REST API request:

```
POST /user/login HTTP/1.1
Host: some.vulnerable.server
Content-Type: application/json
...
```

```
{"username": "lucas.radebe", "password": "Th3Chi3f1992"}
```

The code to build the query may look something like the following:

```
let query = {
  username: req.body.username, // Username read from API request
  password: req.body.password // Password read from API request
}

User.find(query, function(){ ... log user in...}) // Query and authenticate the user.
```

In this example the JSON property values are read from the user submitted object and included within the query. If the user entered valid credentials he/she is authenticated.

However, consider the attacker sends the following request instead:

```
POST /user/login HTTP/1.1
Host: some.vulnerable.server
Content-Type: application/json
...
```

```
{"username": "lucas.radebe", "password": {"$ne": "foobar"}}
```

In this case, the password property has been changed from a simple string to an object containing a comparison operator `{"$ne": "foobar"}`. Now the query is modified to return records where the username is "lucas.radebe" and the password is not equal (`$ne`) to "foobar":

```
User.find({username: "lucas.radebe", password: {"$ne": "foobar"}});
```

Providing the user "lucas.radebe" does not have plaintext password equal to "foobar", the expected password validation process is bypassed and the attacker is authenticated.

The impact of this flaw depends on the query, NoSQL database type and data stored within referenced documents. For example, its typically possible to extract data held within the database by exploiting this flaw.

Note that since JSON objects used within REST API's are compatible with NoSQL queries, its not uncommon to find the developer has built the entire query client-side and then passes it directly into the server side query function.

Injection into the \$where clause

Many NoSQL platforms such as MongoDB support the use of JavaScript within queries via the `$where` clause. In many ways this feature could allow NoSQL queries to feel more natural to a developer who is used to relational SQL databases that make use the WHERE clause.

For example, consider the following Python Flask application, the code below allows users to lookup member information by supplying an "id" value:

```
@app.route("/members")
def lookup_member():
    id = request.args.get("id", None) # id value from the user
    results = db.users.find(
        {
            "$where": "function(){ return obj.id === " + id + "}" # vulnerable
        }
    )
```

The code above is vulnerable since it builds the JavaScript function in insecure way. By including user/attacker controllable input within the JavaScript string, it is possible for the attacker to manipulate the code executed by the server via the `$where` clause. For example, accessing http://vulnerable_server/members?id=1 | evil_code will result in the following code being executed by the server:

```
function(){
    return obj.id === 1 || evil_code
}
```

Depending on the server type `evil_code` could carry out a number of actions including accessing database data, modifying records, shutting the server down or executing system commands.

AppCheck will attempt to safely exploit this flaw to extract a list of document properties and list them within the details section of this finding.

4.2.1. REMEDIATION

When possible user controllable input should not be included within dynamically evaluated code. Instead, an alternative method should be sought to keep data and code separate. If this isn't possible, data should be strictly validated against a whitelist of known good input.

4.2.2. TECHNICAL OVERVIEW

Vulnerable parameters:

App	URI	Parameter
http://192.168.0.70:4000	/graphql	payload.json.variables.book__id

4.2.3. TECHNICAL ANALYSIS

Example: <http://192.168.0.70:4000/graphql> [param: payload.json.variables.book__id]

Exploit

This vulnerability was exploited to provide proof of concept evidence to support the finding. The exploit injects a JavaScript payload to extract properties from the target environment. In this case, the first 10 properties are enumerated from the *obj* object.

Total number of properties: 5

Property Names:

```
_id
name
genre
authorId
__v
```

Technical Details

The vulnerability was detected by injecting JavaScript code designed to trigger a specific time delay from the server.

For example, injecting the following payloads via the **payload.json.variables.book__id** parameter will trigger a 1 and 10 second delay respectively:

```
if(typeof Qgd0J===undefined){var a=new Date();do{var b=new Date();}while(b-a<1000);Qgd0J=1;}
if(typeof 0HzJL===undefined){var a=new Date();do{var b=new Date();}while(b-a<10000);0HzJL=1;}
```

To ensure the best possible accuracy, injected time delays were measured over several high-low cycles.

The log below shows the injected delay and measured server response time:

```
Confirmation cycle 0: Response delay: 10.22 (injected: 10.00)
Confirmation cycle 1: Response delay: 1.16 (injected: 1.00)
Confirmation cycle 2: Response delay: 10.18 (injected: 10.00)
Confirmation cycle 3: Response delay: 10.12 (injected: 10.00)
Confirmation cycle 4: Response delay: 1.12 (injected: 1.00)
Confirmation cycle 5: Response delay: 1.09 (injected: 1.00)
Confirmation cycle 6: Response delay: 1.09 (injected: 1.00)
Confirmation cycle 7: Response delay: 10.10 (injected: 10.00)
```

4.3. INSECURE DIRECT OBJECT REFERENCE (IDOR)



CVSS Score: 6.8 **CVSS Vector:** AV:N/AC:L/Au:S/C:C/I:N/A:N **Impact/Probability:** High/Medium

Affected: http://192.168.0.70:4000

Insecure direct object references (IDOR) is a type of access control vulnerability whereby the attacker is able to access restricted data by manipulating a client supplied identifier. For example, consider an application that loads a users profile information using a URL such as `https://website/profile?user_id=50`. In this example, the supplied `user_id` value directly references the user profile in the backend database. A IDOR vulnerability occurs when the attacker is able to change the supplied value to access another users data, e.g. `https://website/profile?user_id=51`.

IDOR vulnerabilities can occur for a variety of reasons including direct database references, predictable file names and other cases where the attacker is able to manipulate a reference value to bypass access controls.

From OWASP Top 10 2013 (part of Broken Access Control in later versions):

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

Note: This finding may be a false positive if the application permits access to the identified data by design.

References

- <https://www.troyhunt.com/owasp-top-10-for-net-developers-part-4/>
- [https://wiki.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_\(OTG-AUTHZ-004\)](https://wiki.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_(OTG-AUTHZ-004))

4.3.1. REMEDIATION

Ensure proper access controls are enforced to prevent direct access to restricted data.

4.3.2. TECHNICAL OVERVIEW

Vulnerable parameters:

App	URI	Parameter
http://192.168.0.70:4000	/graphql	payload.json.variables.user__id

4.3.3. TECHNICAL ANALYSIS

Example: <http://192.168.0.70:4000/graphql> [param: payload.json.variables.user__id]

Technical Details

AppCheck detected a possible IDOR vulnerability within the target application.

Rationale

Under normal operation the value **102** was submitted via the **payload.json.variables.user__id** parameter. The following potentially sensitive data was identified within the page:

Data Type/Description: Email addresses were found within the page.

Example Data:

nickb@appcheck-ng.com

By changing the **payload.json.variables.user__id** parameter value to **100** the same data type was returned but the values were different

New Data Value:

garyo@appcheck-ng.com, garys_friend@appcheck-ng.com

Other Examples

Modified Value	Data
101	grahamb@appcheck-ng.com , grahams_friend@appcheck-ng.com

4.4. SLOWLORIS DENIAL OF SERVICE (DOS)



CVSS Score: 6.8 **CVSS Vector:** AV:N/AC:M/Au:N/C:P/I:P/A:P **Impact/Probability:** Medium/High

Affected: http://192.168.0.70:4000

Slowloris works by opening multiple connections to the targeted web server and keeping them open as long as possible. It does this by continuously sending partial HTTP requests, none of which are ever completed. The attacked servers open more and connections open, waiting for each of the attack requests to be completed.

Periodically, the Slowloris sends subsequent HTTP headers for each request, but never actually completes the request. Ultimately, the targeted server's maximum concurrent connection pool is filled, and additional (legitimate) connection attempts are denied.

By sending partial, as opposed to malformed, packets, Slowloris can easily slip by traditional Intrusion Detection systems.

Known Affected Configurations

- Apache 1.x, 2.x (CVE-2007-6750)
- Apache Tomcat 7.0.x (CVE-2012-5568)
- All versions prior to Node.js 6.15.0, 8.14.0, 10.14.0 and 11.3.0 (CVE-2018-12122)
- Flask, development mode.

4.4.1. REMEDIATION

While there are no reliable configurations of the affected web servers that will prevent the Slowloris attack, there are ways to mitigate or reduce the impact of such an attack.

In general, these involve increasing the maximum number of clients the server will allow, limiting the number of connections a single IP address is allowed to make, imposing restrictions on the minimum transfer speed a connection is allowed to have, and restricting the length of time a client is allowed to stay connected.

In the Apache web server, a number of modules can be used to limit the damage caused by the Slowloris attack; the Apache modules `mod_limitipconn`, `mod_qos`, `mod_evasive`, `mod_security`, `mod_noloris`, and `mod_antiloris` have all been suggested as means of reducing the likelihood of a successful Slowloris attack. Since Apache 2.2.15, Apache ships the module `mod_reqtimeout` as the official solution supported by the developers.

Other mitigating techniques involve setting up reverse proxies, firewalls, load balancers or content switches.

Administrators could also change the affected web server to software that is unaffected by this form of attack. For example, `lighttpd` and `nginx` do not succumb to this specific attack.

4.4.2. TECHNICAL ANALYSIS

Example: <http://192.168.0.70:4000/>

See Also

<https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris/>

Contents

Test Started: 09/06/2020 - 12:15:02
Test Ended: 09/06/2020 - 12:17:18
Socket timeout appears to be set to 125.026695013 seconds, for the baseline request
Socket timepout was 136.082692862 seconds, for the payload request
Delay in response between the base line request and payload request was 11.0556797981 seconds

Summary

The web server appears to be vulnerable to a Slowloris DoS attack

4.5. POSSIBLE SENSITIVE DATA DISCLOSURE (CREDENTIALS)



CVSS Score: 5.0 **CVSS Vector:** AV:N/AC:L/Au:N/C:P/I:N/A:N **Impact/Probability:** Info/Medium

Affected: http://192.168.0.70:4000

The application appears to disclose sensitive data within the servers response. AppCheck analyses responses and applies pattern matching rules to identify potential data disclosures. You should review this finding to determine if the data is sensitive and erroneously disclosed.

4.5.1. REMEDIATION

You should review the disclosed data to determine if it is sensitive and accessible in a way that should be restricted and if so take appropriate steps to secure the data.

4.5.2. TECHNICAL ANALYSIS

Example: <http://192.168.0.70:4000/graphql>

Technical Details

The JSON response appears to include sensitive properties (Matching **password**).

Matching Data

```
{"data":{"user":{"id":102,"email":"nickb@appcheck-ng.com","password":"70ccd9007338d6d81dd3b6271621b9cf9a97ea00","first_name":"Dr","last_name":"Nick","friends":[]}}}
```

4.6. GRAPHQL INTROSPECTION SUCCESSFUL



Impact/Probability: Info/Info

Affected: http://192.168.0.70:4000

A GraphQL schema has a built-in introspection system that publishes the schema's structure. AppCheck attempts to enumerate all available queries via introspection so that they can be audited for security flaws. This finding is emitted when a GraphQL endpoint is detected and introspection is attempted.

4.6.1. REMEDIATION

This finding is presented as informational.

4.6.2. TECHNICAL ANALYSIS

Example: <http://192.168.0.70:4000/graphql>

Technical Details

The following GraphQL queries, mutations and subscriptions were enumerated via introspection:

GraphQL Query 0.

Query:

```
query RootQueryType($author__id: ID, $book__id: ID){
  book(id: $book__id){
    id
    name
    genre
    author(id: $author__id){
      id
      name
      age
      books{
        id
        name
        genre
      }
    }
  }
}
```

Variables:

```
{"book__id": "5edf5e5bdb63a75dc07ad1fGARY", "author__id": "5edf7a06951a931accf1555b"}
```

GraphQL Query 1.

Query:

```
query RootQueryType($author__id: ID){
  author(id: $author__id){
    id
    name
    age
    books{
      id
      name
      genre
      author(id: $author__id){
        id
        name
        age
      }
    }
  }
}
```

Variables:

```
{"author__id": "5edf7a06951a931accf1555b"}
```

GraphQL Query 2.

Query:

```
query RootQueryType($author__id: ID){
  books{
    id
    name
    genre
    author(id: $author__id){
      id
      name
      age
      books{
        id
        name
        genre
      }
    }
  }
}
```

Variables:

```
{"author__id": "5edf7a06951a931accf1555b"}
```

GraphQL Query 3.

Query:

```
query RootQueryType($user__id: Int){
  user(id: $user__id){
    id
    email
    password
    first_name
    last_name
  }
}
```

```
    friends{
      id
      email
      password
      first_name
      last_name
    }
  }
}
```

Variables:

```
{"user__id": 102}
```

GraphQL Query 4.

Query:

```
query RootQueryType{
  users{
    id
    email
    password
    first_name
    last_name
    friends{
      id
      email
      password
      first_name
      last_name
    }
  }
}
```

Variables:

```
{}
```

GraphQL Query 5.

Query:

```
query RootQueryType($author__id: ID){
  authors{
    id
    name
    age
    books{
      id
      name
      genre
      author(id: $author__id){
        id
        name
        age
      }
    }
  }
}
```

Variables:

```
{"author__id": "5edf7a06951a931accf1555b"}
```

GraphQL Query 6.

Query:

```
mutation Mutation($author__id: ID!, $addAuthor__name: String!, $addAuthor__age: Int!){
  addAuthor(name: $addAuthor__name, age: $addAuthor__age){
    id
    name
    age
    books{
      id
      name
      genre
      author(id: $author__id){
        id
        name
        age
      }
    }
  }
}
```

Variables:

```
{"addAuthor__age": 100, "addAuthor__name": "daoqvjnp", "author__id": "5edf7a06951a931accf1555b"}
```

GraphQL Query 7.

Query:

```
mutation Mutation($author__id: ID!, $addBook__name: String!, $addBook__genre: String!, $addBook__authorId: ID){
  addBook(name: $addBook__name, genre: $addBook__genre, authorId: $addBook__authorId){
    id
    name
    genre
    author(id: $author__id){
      id
      name
      age
      books{
        id
        name
        genre
      }
    }
  }
}
```

Variables:

```
{"addBook__name": "ka1cjdne", "addBook__authorId": "osqtkfvb", "author__id": "5edf7a06951a931accf1555b", "addBook__genre": "bzfzjzzy"}
```

4.7. GRAPHQL ENDPOINT DETECTED



Impact/Probability: Info/Info

Affected: http://192.168.0.70:4000

A GraphQL endpoint was detected. AppCheck will attempt to enumerate and test GraphQL queries for vulnerabilities.

4.7.1. REMEDIATION

This finding is presented as informational.

4.7.2. TECHNICAL ANALYSIS

Example: Messages

--->

```
POST /graphql HTTP/1.1
Content-Length: 133
Host: 192.168.0.70:4000
Accept: application/json
Connection: close
Referer: http://192.168.0.70:4000/
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36
Content-Type: application/json
Accept-Encoding: Identity
```

```
{"query": "query detect($name: String!){\n\t__type(name:$name){\n\t name\n\t description\n\t}\n}", "variables": {"name": "String"}}
```

<---

```
HTTP/1.1 200 OK
Content-Length: 235
X-Powered-By: Express
Date: Tue, 09 Jun 2020 12:01:09 GMT
```

Connection: close

Etag: W/"eb-yRFMnpxgra7U0vsHSyPsU/9xaRI"

Content-Type: application/json; charset=utf-8

Access-Control-Allow-Origin: *

```
{"data":{"__type":{"name":"String","description":"The `String` scalar type represents textual data, represented as UTF-8 character sequences. The String type is most often used by GraphQL to represent free-form human-readable text."}}}
```

5. EXTENDED INFORMATION

5.1. PORT SCAN

Please review the following open ports. You should ensure there are no unnecessary ports or services open.

Host	Port	Service	Version
192.168.0.70	4000	http	

6. APPENDIX A: WEB APPLICATION DEFENSIVE STRATEGIES

6.1. VALIDATING INPUT

Input validation is an important defence strategy in preventing SQL injection, code injection, cross site scripting and a host of other vulnerabilities. The aim of input validation is to ensure that data processed by the application contains only known good characters and that all others are discarded.

There are two high approaches that can be adopted to achieve this aim: white listing and black listing. It is widely accepted that the white list approach is the most secure option. However, where possible a two tiered approach to input validation is beneficial.

Black Listing: Reject Known Bad

Note: Black listing should not be used in isolation but as a complementary measure to white listing. Data that successfully passes the black list filter should then be validated using the white list approach.

Black listing is the process of rejecting known bad input. This technique receives a lot of criticism from security purists because it is impossible to know all dangerous character sequences ahead of time. Whilst this is true, a black list approach can be a good first line of defence against attempts to discover and exploit vulnerabilities within the application. Any user violates the black list filter should have any active sessions invalidated (logged out) and the event should be logged. In some circumstances an alerting process may be appropriate, depending on the sensitivity of the chosen black list and the expected volume of false positives.

- Create a black list containing characters and character sequences which are associated with the attack or attacks.
- Decode all input until no further decoding is possible before comparing against the black list.
- Users who violate the filter should have their session invalidated and a logging and/or alerting process should be invoked.
- Black listing should only be used as a complementary measure to white listing. Data that successfully passes the black list filter should then be passed to the white list process.

White Listing: Allow Known Good

White listing is the process of validating input to ensure it contains only known good (safe) characters and is constructed in an expected manner. This approach is preferred over black listing since it does not require that the developer know all potentially dangerous characters ahead of time. A white list filter should be context dependant and be as restrictive as possible. Potential features to validate include:

- Data contains only permitted, safe characters.
- The data is of a certain length (e.g. falls between a minimum and maximum length).
- The data matches a defined regular expression (e.g. to validate the structure of an email address).

Care should be taken to ensure the white list is sufficiently restrictive to be secure whilst allowing the user to include all of the characters he/she needs to interact with the application.

Sanitising (Converting) Output

Whenever data that originated from a user or an out of bound channel is copied into a servers' response it should be HTML encoded to sanitise potentially dangerous characters. HTML encoding is the process of converting characters to their HTML equivalents. The converted character appears the same as the original character when viewed in a web browser but does not affect the structure of the HTML document.

The recommended approach is to HTML encode all non-alphanumeric characters to ensure that all special characters that could be useful in constructing a malicious script are converted.

The following characters should always be converted to their inert HTML equivalents before being included within the page:

Character	HTML Equivalent
"	"e;
'	'
&	&
<	<
>	>

Additional characters can be converted to HTML equivalents using their character code in decimal prefixed with `&#` and terminated with a semicolon as follows:

Character	HTML Equivalent
;	;
+	+

XSS: Remove Unnecessary Exploit Vectors.

Performing input validation and output sanitisation will form a robust defence against reflected and stored cross site scripting vulnerabilities. However there are a number of circumstances where cross site scripting vulnerabilities may still be possible. There are a number of locations where it is inherently dangerous to insert user supplied data.

Where possible user supplied data should not be included within existing scripts. Doing so increases the chances that an input validation filter can be bypassed since the attacker does not need as many special characters such as `<` and `>` to construct a malicious script.

Enforce Password Complexity

Enforcing minimum password strength can greatly improve authentication security, especially when combined with an account lockout policy.

Factors to consider are:

- Minimum Password Length (8 characters and above is industry standard).
- Enforcing mixed case characters.
- Mandatory requirement for numeric and special characters.
- Reject attempts to configure a password that is the same as or derived from the username.
- Prevent users from configuring a common weak password by comparing it to a word list of known weak passwords.

The benefit of adopting all of the above restrictions is that users are forced into selecting a secure password that cannot be easily cracked via a remote dictionary/brute force attack. The downside to this approach is that users are more likely to write a password down if it is difficult to remember.

Prevent Dictionary / Brute Force Attacks

There are a number of approaches to prevent or slow down dictionary and brute force attacks against an authentication system. One of the most effective approaches is to disable user accounts following a defined number of failed authentication attempts. The account should remain disabled until either an administrator or the legitimate user re-enables the user account. Some applications simply lock the account for a predefined time period such as 30 minutes to slow down the dictionary attack.

The following account lockout policies offer some potential scenarios to consider:

- **Lockout After X Bad Attempts: Administrator Re-Enable**
This approach requires an administrator or moderator to re-enable locked accounts. This approach is typically only viable for applications that have a relatively small number of users. Applications with thousands of users who most likely generate more than an acceptable number of account related support requests. The main benefit of this system is that the administrator is able to investigate the reason for the account lockout. Systems that require a high level of security such as internet banking should employ this system (along with external written or phone verification).
- **Lockout After X Bad Attempts: User Re-Enable**
An effective method of enforcing account lockouts without incurring the administrative overhead is to allow the legitimate user to re-enable their own user account. The process to re-enable the account should be sufficiently secure; one approach is to send the user an email containing a time limited link to their registered email address. The link should direct the user to a portion of the application that permits them to answer a security question which will then reset the password. This approach does have security risks associated with it. Firstly, email could be intercepted by a malicious third party who may then attempt to exploit the password reset process. However, providing the validation process is suitably secure, the system should offer a high enough level of security for most applications.
- **Lockout after X Bad Attempts: Re-Enable Following X Minutes**

A less secure option than the previously described methods is to suspend a user account for a predefined time period (e.g. 30 mins). Once the time period has expired the account is re-enabled. This method will slow down an attempt to guess passwords via a dictionary attack but may not prevent it.

Implement a CAPTCHA System

A CAPTCHA (“Completely Automated Public Turing test to tell Computers and Humans Apart”) is a type of challenge response system where the user is asked to enter numbers and letters that appear within an image displayed by the web application, in order to prevent automated attacks against the authentication systems. This method can provide an excellent level of security without increasing the administrative overhead. Some research into CAPTCHA systems has demonstrated that automated attacks are possible under certain circumstances, however the effort required to attack such a system is great enough to deter the majority of attackers.

Validate Credentials Properly

Ensure that your username and password verification code is secure.

- Allow a wide range of characters within the username and password.
- Perform case sensitive validation.
- Allow usernames and passwords to be up to 20 characters in length.

Prevent Information Disclosure

Use a standard error message for authentication attempts that does not disclose which component failed authentication.

Do not disclose whether or not a username is valid within password recovery functionality. For example, use the following message instead. “You will receive an email allowing you reset your password if the account entered is valid on the system”.

Prevent Misuse of Password Change Functionality

Password change functionality should be restricted to authenticated sessions. Users should be required to enter their old password as well as the new password. All username information should be stored server side and the client should not be allowed to specify a username within this process.