</talentlabs>

Express Lecture 13

SQL Injection and XSS in Express

</talentlabs>

Agenda

- SQL Injection
- XSS (Cross-site-scripting)

</talentlabs>

Introduction

Security is a huge part in backend development. Using standard frameworks (correctly) is the first step to make sure your application safe.

SQL injection and XSS are the most fundamental security issues therefore we would like you to have some hands on experience about them.

Review

SQL Injection

What is SQL injection?

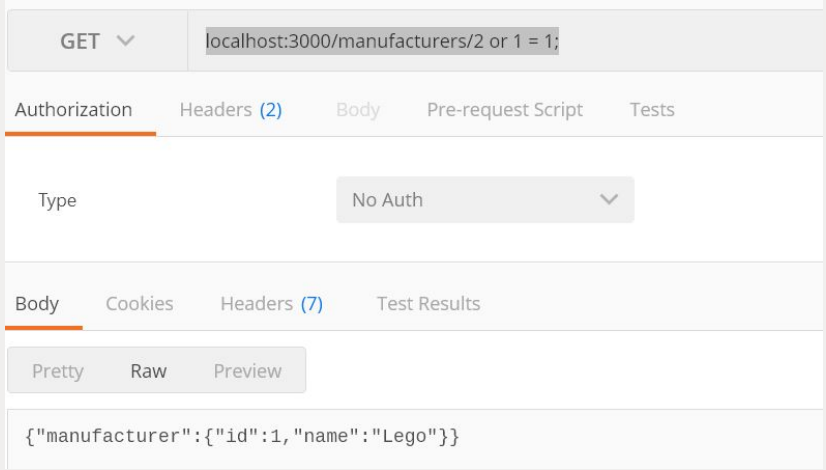
- SQL injection is a code injection technique that might destroy your database.
- SQL injection is one of the most common web hacking techniques.
- SQL injection is **the placement of malicious code in SQL statements, via web page input.**

Bad Express code

```
1  /* Retrieve a manufacturer with id = :id */
2  router.get('/manufacturers/:id', function(req, res, next) {
3    //knex connection
4    connection
5      .raw(`select * from manufacturer where id = ` +
req.params["id"])
6      .then(function (result) {
7        var manufacturers = result[0];
8        // send back the query result as json
9        res.json({
10         manufacturer: manufacturers[0],
11       });
12     })
13     .catch(function (error) {
14       // log the error
15       console.log(error);
16       res.json(500, {
17         message: error,
18       });
19     });
20 });
```

Naive string concatenation….

localhost:3000/manufacturers/2 or 1 = 1;

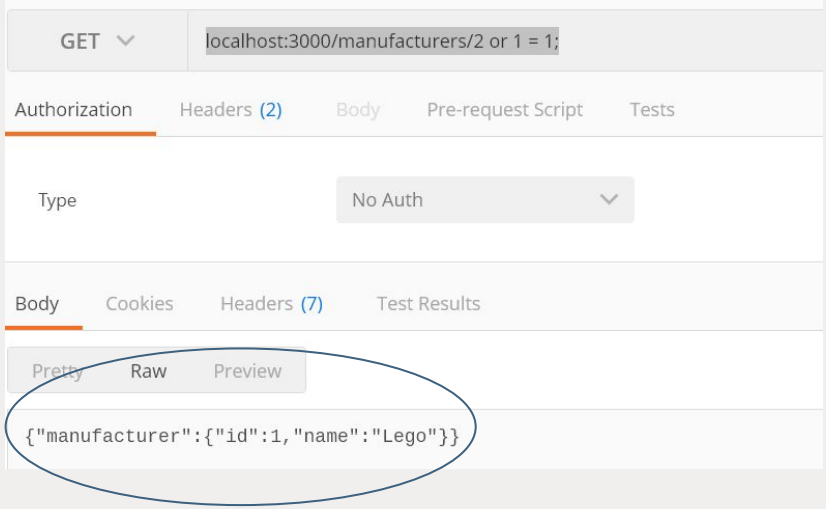


Bad Express code (Leaking data demo)

```
1  /* Retrieve a manufacturer with id = :id */
2  router.get('/manufacturers/:id', function(req, res, next) {
3    //knex connection
4    connection
5      .raw(`select * from manufacturer where id = ` +
req.params["id"])
6      .then(function (result) {
7        var manufacturers = result[0];
8        // send back the query result as json
9        res.json({
10         manufacturer: manufacturers[0],
11       });
12     })
13     .catch(function (error) {
14       // log the error
15       console.log(error);
16       res.json(500, {
17         message: error,
18       });
19     });
20 });
```

Naive string concatenation….

localhost:3000/manufacturers/2 **or 1 = 1;**



Requested id=2 but got id=1 (leaking data)

Bad Express code (Dropping data demo)

```
1  /* Retrieve a manufacturer with id = :id */
2  router.get('/manufacturers/:id', function(req, res, next) {
3    //knex connection
4    connection
5      .raw(`select * from manufacturer where id = ` +
req.params["id"])
6      .then(function (result) {
7        var manufacturers = result[0];
8        // send back the query result as json
9        res.json({
10         manufacturer: manufacturers[0],
11       });
12     })
13     .catch(function (error) {
14       // log the error
15       console.log(error);
16       res.json(500, {
17         message: error,
18       });
19     });
20 });
```

Naive string concatenation….

localhost:3000/manufacturers/1;**delete from product;delete from manufacturer;**

Error: ER_PARSE_ERROR: You have an error in your SQL syntax;

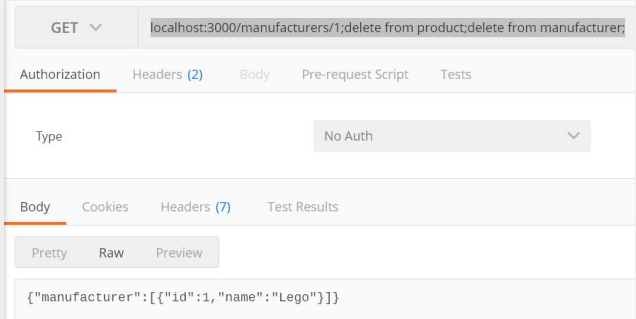
This doesn't work! Why?
Because **most database connectors disabled multiple SQL statements execution** by default! (because it is dangerous).

Bad Express code (Dropping data demo)

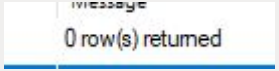
```
development: {
  client: "mysql",
  connection: {
    host: "student-mysql.ccttwiegufhh.us-east-2.rds.amazonaws.com",
    user: "studentmysql",
    password: "studentmysql",
    database: "express_lecture",
    multipleStatements: true,
  },
},
```

Let's enable multipleStatements in ./knexfile.js!

localhost:3000/manufacturers/1;**delete from product;delete from manufacturer;**



With MySQL Workbench:

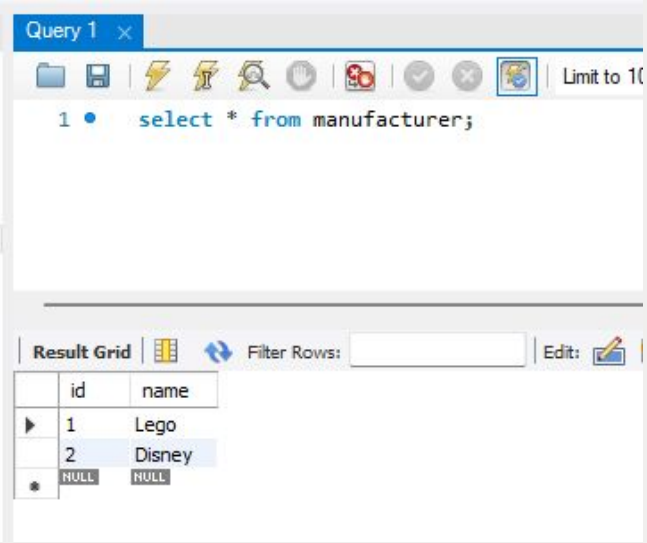


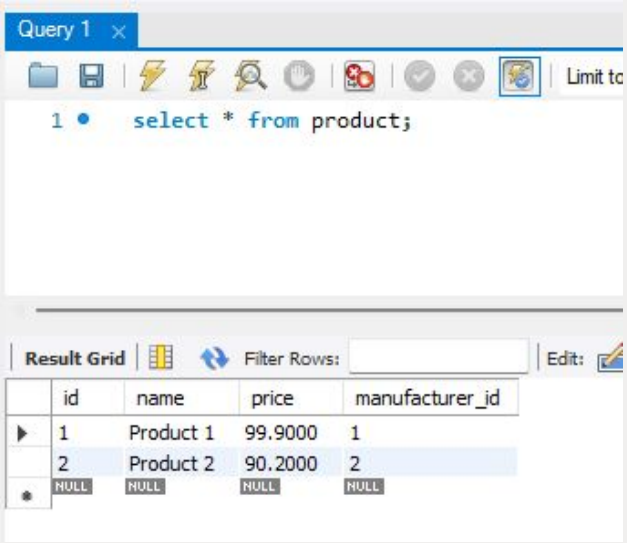
Execute the seed file to get back the deleted data

Run these to execute the 2 seed files we created:

```
npx knex seed:run --specific=initial-manufacturer.js
npx knex seed:run --specific=initial-product.js
```

Check in MySQL Workbench





Parameter binding

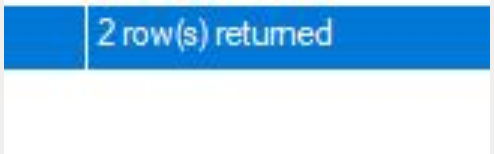
```
router.get('/manufacturers/:id', function(req, res, next) {
  //knex connection
  connection
    .raw(`select * from manufacturer where id = ?`, [req.params["id"]])
    .then(function (result) {
      var manufacturers = result[0];
      // send back the query result as json
      res.json({
        manufacturer: manufacturers[0],
      });
    })
    .catch(function (error) {
      // log the error
      console.log(error);
      res.json(500, {
        "message": error
      });
    });
});
```

This way we are telling the database:
We only have 1 value to be put here!!

Try this again:

localhost:3000/manufacturers/1;delete from product;delete from manufacturer;

The issue is migrated!



Summary - To avoid SQL injection.

1. Don’t allow the app to run multiple statements in 1 shot.
2. Use Parameter binding!

XSS (Cross-site-scripting)

XSS

2 steps:

1. Hacker **write JavaScript into your database**.
2. Your website **display those JavaScript code directly** without protection.

JavaScript code got inserted into the database

localhost:3000/manuf

POST localhost:3000/manufacturers/

Authorization Headers (2) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
1 {
2   "name": "<script>alert('Hacked!')</script>"
3 }
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview

```
{"message":"Done"}
```

Result Grid Filter Rows:

id	name
1	Lego
2	Disney
3	<script>alert('Hacked!')</script>
NULL	NULL

manufacturer 3

localhost:3000/manuf No Environment

GET localhost:3000/manufacturers/ Params Send Save

Authorization Headers (2) Body Pre-request Script Tests Code

Type No Auth

Body Cookies Headers (7) Test Results Status: 200 OK Time: 239 ms

Pretty Raw Preview


```
{"manufacturers":[{"id":1,"name":"Lego"},{"id":2,"name":"Disney"},{"id":3,"name":"<script>alert('Hacked!')</script>"}]}
```

Prepare the victim HTML page

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet'
href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

Result Grid | Filter Rows:

id	name
1	Lego
2	Disney
3	<script>alert('Hacked!')</script>
NULL	NULL

manufacturer 3

```
router.get('/', function(req, res, next) {
  //knex connection
  connection
    .raw(`select * from manufacturer where id
= ?`, [3])
    .then(function (result) {
      var manufacturers = result[0];
      // send back the query result as json
      res.render('index', {
        title: manufacturers[0].name,
      });
    })
    .catch(function (error) {
      // log the error
      console.log(error);
      res.json(500, {
        message: error,
      });
    });
})
```

It doesn’t work

<script>alert('Hacked!')</script>

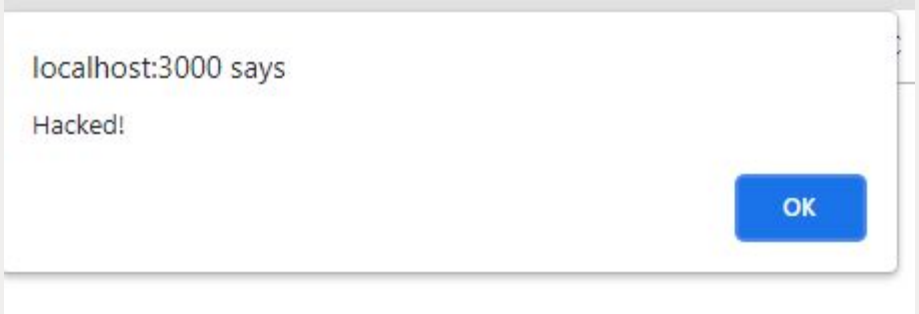
Welcome to <script>alert('Hacked!')</script>

In EJS template <%= %> tag will escape the value (make it to be text instead of a HTML script tag)

Unsafe evaluate

There are some situations that we want to store the data as HTML and display them directly. For example, a Blog post.
We can do it with <%- %> tag in EJS.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet'
href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%- title %></h1>
    <p>Welcome to <%- title %></p>
  </body>
</html>
```



The JavaScript code in the <script> tag got executed!

Unsafe evaluate + XSS filter

There are some situations that we want to store the data as HTML and display them directly. For example, a Blog post.
We can do it with <%- %> tag in EJS.

Let's say we are working on a Blog website and we only want user to have safe HTML tags in their blog post. **We can do it by whitelisting HTML tags with XSS filter**

https://www.npmjs.com/package//sanitize-html

npm install sanitize-html

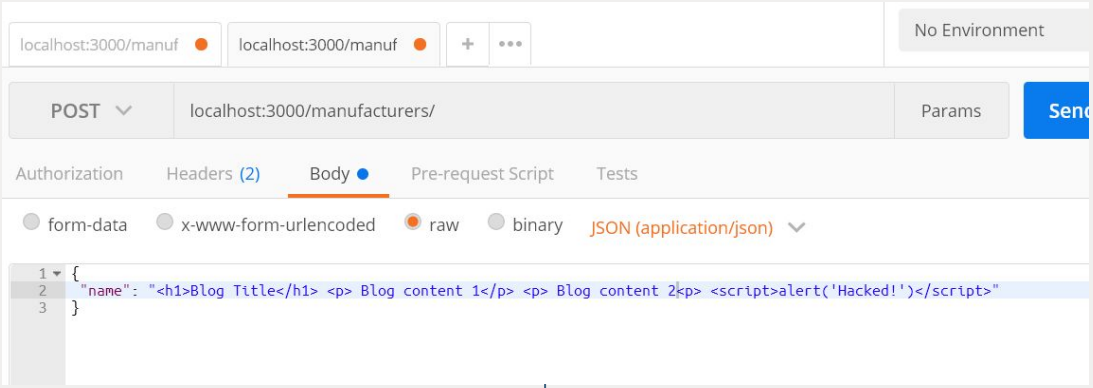
Unsafe evaluate + XSS filter

```
router.post('/manufacturers', function(req, res, next) {
  console.log("POST Request", req.body);
  var promise = connection.raw(
    `
    insert into manufacturer (name)
    values (?)
    `,
    [sanitizeHtml(req.body["name"])]
  );
  promise.then(function (result) {
    res.json({
      "message": "Done",
    })
  }).catch(function (error) {
    // log the error
    console.log(error);
    res.json(500, {
      message: error,
    });
  });
});
```

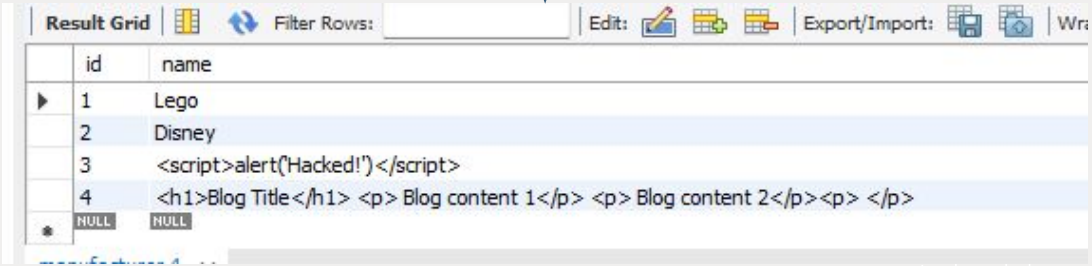
Let's say we are working on a Blog website and we only want user to have safe HTML tags in their blog post. **We can do it by whitelisting HTML tags with XSS filter**

Unsafe evaluate + XSS filter

```
router.post('/manufacturers', function(req, res, next) {
  console.log("POST Request", req.body);
  var promise = connection.raw(
    `
    insert into manufacturer (name)
    values (?)
    `,
    [sanitizeHtml(req.body["name"])]
  );
  promise.then(function (result) {
    res.json({
      "message": "Done",
    })
  }).catch(function (error) {
    // log the error
    console.log(error);
    res.json(500, {
      message: error,
    });
  });
});
```



Migrated!



Summary - To avoid XSS

Frontend:
Most modern frontend frameworks will handle this for you. For example, react will not render some dangerous HTML tags directly, like the <script> tag.

Backend:
We can apply XSS filters to remove dangerous content before it got into our database. There are many filters pick the one that fit your application.

https://www.npmjs.com/package//sanitize-html
https://www.npmjs.com/package/xss-filters