NCS: Lab 4 - Web Security 2

Maruf Asatullaev m.asatullaev@innopolis.university

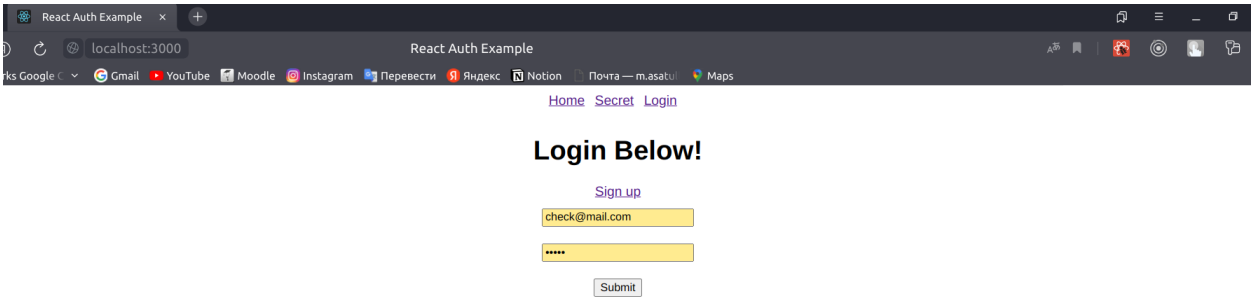
Nataliya Sakovets n.sakovets@innopolis.university

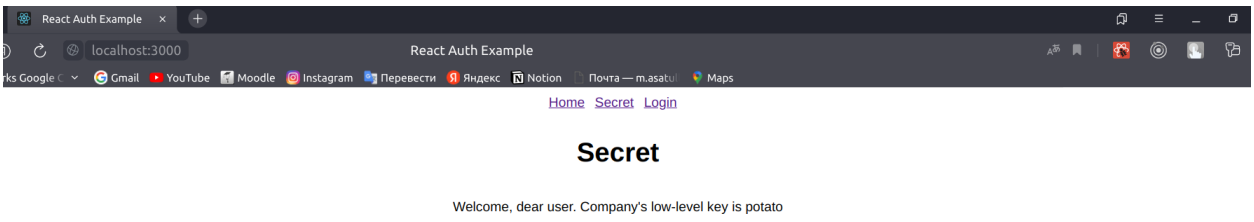
Dmitriy Pirozhenko d.pirozhenko@innopolis.university

Karina Tyulebaeva k.tyulebaeva@innopolis.university

Overview

It is ordinary authentification app, which supports user registration, gives secret key for authentified users depending on their role





Broken Access Control

Description

Broken Access Control is a type of vulnerability in which an attacker can gain unauthorized access to resources or functionality that they are not supposed to have access to. This can occur when access controls, such as authentication or authorization mechanisms, are not properly implemented or enforced. It can lead to sensitive data being exposed, user accounts being compromised, and other security issues.

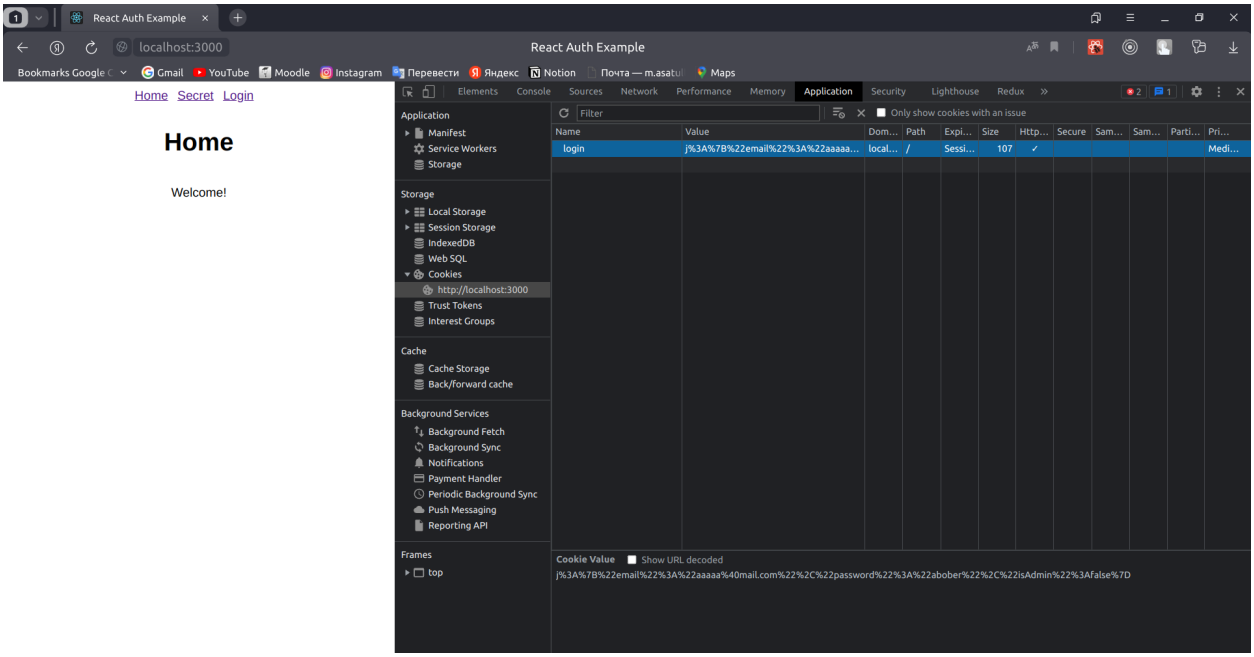
Vulnerability

While logging in we are making **api/authenticate** request and is we found email in database, then we check password correctness:

```
93      } else {
94        user.isCorrectPassword(password, function(err, same) {
95          if (err) {
96            res.status(500)
97              .json({
98              error: 'Internal error please try again'
99            });
100         } else if (!same) {
101           res.status(401)
102             .json({
103             error: 'Incorrect email or password'
104           });
105         } else {
106           res.cookie('login', req.body, { httpOnly: true }).sendStatus(200);
107         }
108       });
109     }
110   });
111 });
```

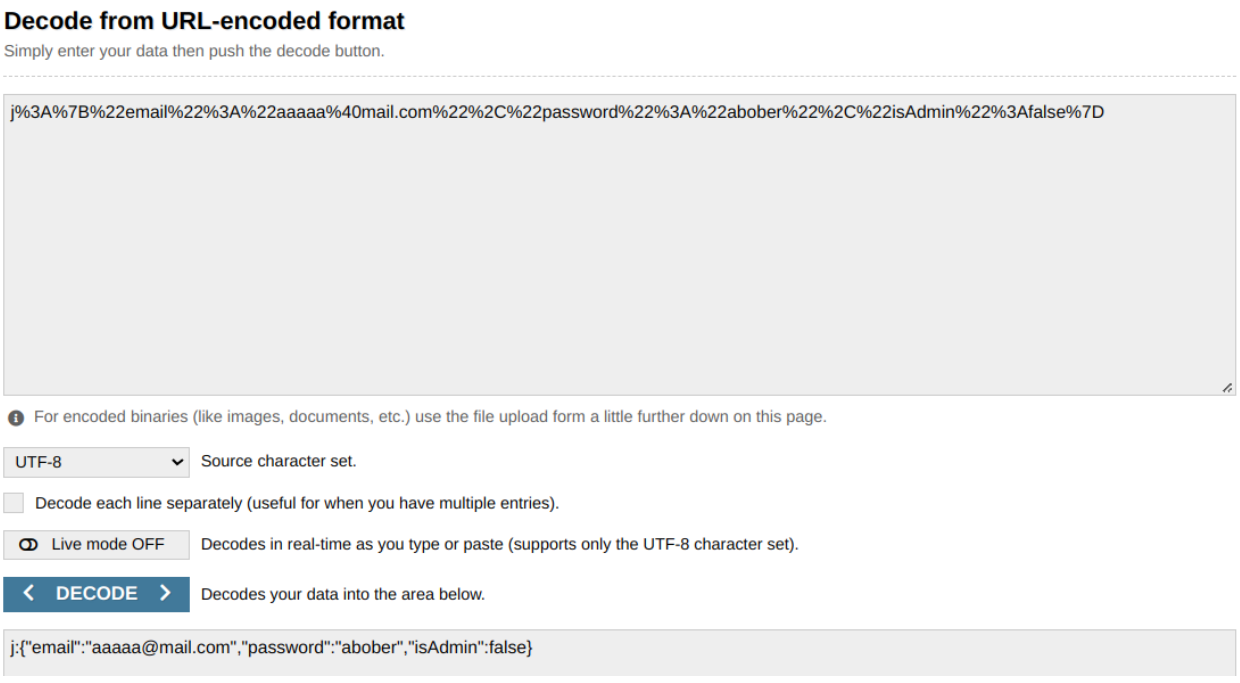
And after confirming successful login we store request data in cookies in order to stay logged in, but we are not encrypting it which makes easy for the attacker to get our login and password

Exploitation



First, we open browser devtools → Application → Cookies. Then we check the value of *login* cookie.

Lastly, we decode that value:

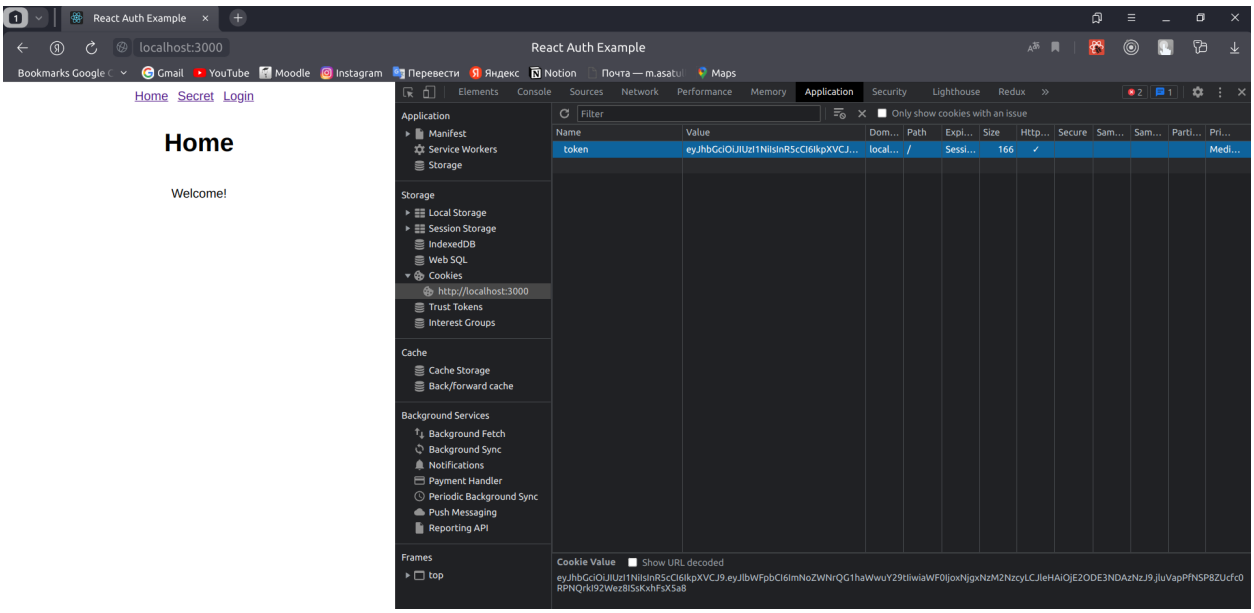


How to fix

Solution here would be using JSON web tokens and also use only login of the user instead of all fields. Let's rewrite that else block with cookie setting this way:

```
102         } else {
103           const payload = { email };
104           const token = jwt.sign(payload, secret, {
105             expiresIn: '1h'
106           });
107           res.cookie('token', token, { httpOnly: true }).sendStatus(200);
108         }
109       });
```

And now we see that all our data is encrypted:



Privilege escalation

Description

Privilege escalation vulnerability is a type of security flaw that allows an attacker to gain higher-level privileges on a system or application than they are supposed to have. This can occur when a system or application fails to properly restrict access to certain

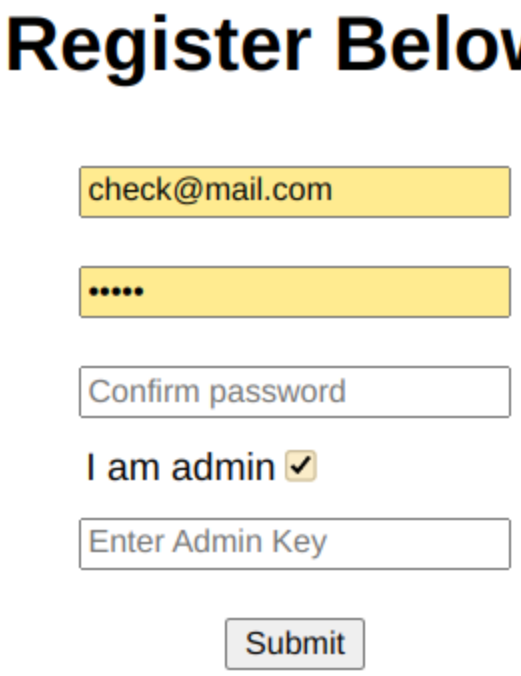
resources or functionality, or when it has weak authentication or authorization mechanisms.

For example, an attacker with limited privileges on a system may be able to exploit a vulnerability in the system to gain administrative privileges, giving them access to sensitive data, control over the system, and the ability to install malicious software.

Privilege escalation vulnerabilities are particularly dangerous because they allow attackers to bypass security controls and gain access to resources that are normally restricted. They can be difficult to detect and mitigate, as they often require a deep understanding of the underlying system architecture and security mechanisms. Organizations can protect against privilege escalation vulnerabilities by implementing strong access controls and regularly patching and updating their systems and applications.

Vulnerability

While signing up, user could choose to be admin, but he/she should provide special admin key.



And in **api/register** request we assigning adminSecretKey to the beginning of email and password to differentiate admins and ordinary users

```
52 app.post('/api/register', function(req, res) {
53   const { adminKey } = req.body;
54   let { email, password } = req.body
55   let isAdmin = false
56
57   if (adminKey === adminSecretKey) {
58     email = `${adminSecretKey}_${email}`
59     password = `${adminSecretKey}_${password}`
60     isAdmin = true
61   }
```

In the future while logging in (**api/authenticate** request) user can choose checkbox and it would be checked in database whether we have such admin email (including **adminSecretKey**):

```
70 app.post('/api/authenticate', function(req, res) {
71   const { isAdmin } = req.body;
72   let { email, password } = req.body;
73   if (isAdmin) {
74     email = `${adminSecretKey}_${email}`
75     password = `${adminSecretKey}_${password}`
76   }
77
78   User.findOne({ email }, function(err, user) {
```

Then in **api/secret** request we split email to define whether it is admin:

```
70  app.post('/api/authenticate', function(req, res) {
71    const { isAdmin } = req.body;
72    let { email, password } = req.body;
73    if (isAdmin) {
74      email = `${adminSecretKey}_${email}`
75      password = `${adminSecretKey}_${password}`
76    }
77
78    User.findOne({ email }, function(err, user) {
```

And in api/secret request we show apropriate info depending on role of logged in user:

```
39  app.get('/api/secret', withAuth, function(req, res) {
40    const adminKey = req.email.split('_')[0]
41
42    if(adminKey === adminSecretKey) {
43      res.send("Welcome, dear administrator. Company's high-level master key is
44    } else {
45      res.send("Welcome, dear user. Company's low-level key is potato");
46    }
47  });
48
```

However this way we storing many duplicates of our **adminSecretKey** in database that could give attacker chance to understand how to become admin

Exploitation

Using mongoose lib we can write small script to connect to our database and by social engineering try to brute force some appropriate names for mogoose.Schema proper name and access our MongoDB

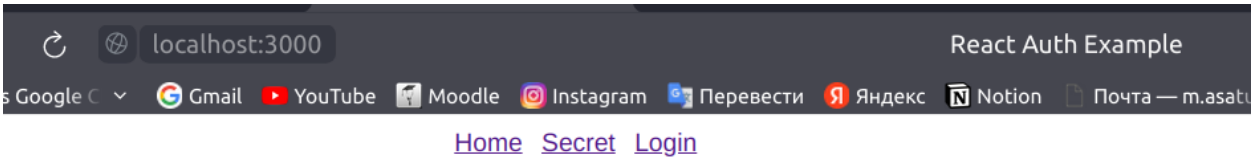
```
JS sqli.js > ...
1   const mongoose = require('mongoose');
2
3   const mongo_uri = 'mongodb://localhost/react-auth';
4   mongoose.connect(mongo_uri, { useNewUrlParser: true, useUnifiedTopology: true });
5
6   const itemSchema = new mongoose.Schema({});
7
8   const pupularNames = ['admin', 'model', 'schema', 'user', 'employee', 'person', 'userAdmin', 'modelSchema'
9
10  pupularNames.forEach(popularName => {
11    const Item = mongoose.model(popularName, itemSchema);
12
13    Item.find({}, (err, docs) => {
14      if (err) {
15        console.log(err);
16      } else {
17        console.log(docs);
18      }
19    });
20  })
21
22
```

So we can see the entries:

```
maruf@lenovo-maruf:~/university/ncs/react-auth-example(master)$ node sqli.js
[]
[]
[]
[
  {
    _id: 643e3aa5bcbc7c6a1bfbb40d,
    email: 'mumble_fan@mail.com',
    password: 'mumble_fan',
    isAdmin: true,
    __v: 0
  },
  {
    _id: 643e3babbcbc7c6a1bfbb416,
    email: 'check@mail.com',
    password: '12345',
    isAdmin: false,
    __v: 0
```

We can make conclusion that users having **isAdmin** field *true* has same name at the beginning of email/password, so the user, for example, can just create such account to get admin data in **api/secret** request:

```
{
  _id: 643e3aa5bcbc7c6a1bfbb40d,
  email: 'mumble_fan@mail.com',
  password: 'mumble_fan',
  isAdmin: true,
  __v: 0
},
{
  _id: 643e3babbcbc7c6a1bfbb416,
  email: 'check@mail.com',
  password: '12345',
  isAdmin: false,
  __v: 0
},
{
  _id: 643e4c60bcbc7c6a1bfbb41c,
  email: 'mumble_boy@mail.com',
  password: 'mumble_girl',
  isAdmin: false,
  __v: 0
}
```

How to fix

Let’s verify admin rights by appropriate isAdmin field and not store it in DB. Firstly we should pass the isAdmin field in **api/authenticate**:

```
94          } else {
95            const isAdmin = user.isAdmin
96            const payload = { email, isAdmin };
97            const token = jwt.sign(payload, secret, {
98              expiresIn: '1h'
99            });
100           res.cookie('token', token, { httpOnly: true }).sendStatus(200);
101         }
102       });
```

Then in middleware:

```
17        } else {
18          req.email = decoded.email;
19          req.isAdmin = decoded.isAdmin;
20          next();
21        }
```

And finnally modify **api/secret**

```
39  app.get('/api/secret', withAuth, function(req, res) {
40    if(req.isAdmin) {
41      res.send("Welcome, dear administrator. Company's high-level master k
42    } else {
43      res.send("Welcome, dear user. Company's low-level key is potato");
44    }
45  });
```

And finally just remove secretAdminKey assigning in **api/register** and **api/authenticate,** so our db would look like this:

Security Misconfiguration (MongoDB insecure)

Description

Security Misconfiguration vulnerability refers to the failure of properly configuring the security settings of a system, application, or network. It occurs when security controls are not implemented, or they are not configured correctly, leaving the system vulnerable to attacks. This vulnerability can be caused by various factors such as using default passwords, leaving unnecessary ports open, not updating software, and not properly securing sensitive data. Attackers can exploit this vulnerability to gain unauthorized access to the system, steal sensitive data, or launch other attacks. To prevent security misconfiguration vulnerability, it is essential to implement and maintain proper security controls, regularly update software and configurations, and follow security best practices.

Vulnerability

As described above during exploitation, attacker could has access to our database which is not okay at all. Also it is bad security practe to store passwords without encrypting them.

Exploitation

Same as we did in previous case. Create empty model, use find() function of mongoose model and try to find appropriate name for model by brute forcing:

```
JS sqli.js > ...
1   const mongoose = require('mongoose');
2
3   const mongo_uri = 'mongodb://localhost/react-auth';
4   mongoose.connect(mongo_uri, { useNewUrlParser: true, useUnifiedTopology: true });
5
6   const itemSchema = new mongoose.Schema({});
7
8   const pupularNames = ['admin', 'model', 'schema', 'user', 'employee', 'person', 'userAdmin', 'modelSchema'
9
10  pupularNames.forEach(popularName => {
11    const Item = mongoose.model(popularName, itemSchema);
12
13    Item.find({}, (err, docs) => {
14      if (err) {
15        console.log(err);
16      } else {
17        console.log(docs);
18      }
19    });
20  })
21
22
```

Then run the script and get all users info:

```
maruf@lenovo-maruf:~/university/ncs/react-auth-example(master)$ node sqli.js
[]
[]
[]
[
  {
    _id: 643e66155570f2a8e98c6a87,
    email: 'check@mail.com',
    password: '12345',
    isAdmin: false,
    __v: 0
  },
  {
    _id: 643e66285570f2a8e98c6a89,
    email: 'bob@mail.com',
    password: 'elen',
    isAdmin: true,
    __v: 0
  },
  {
    _id: 643e66455570f2a8e98c6a8b,
    email: 'jambo23@mail.com',
    password: 'jambo34@mail.com',
    isAdmin: false,
    __v: 0
  }
```

How to fix

Firstly, let's use bcrypt lib (more specifically **bcrypt.hash** function) for hashing our passwords before saving to database:

```
12  UserSchema.pre('save', function(next) {
13    // next()
14    if (this.isNew || this.isModified('password')) {
15      const document = this;
16      bcrypt.hash(this.password, saltRounds, function(err, hashedPassword) {
17        if (err) {
18          next(err);
19        } else {
20          document.password = hashedPassword;
21          next();
22        }
23      });
24    } else {
25      next();
26    }
27  });
```

Commented code is what was before

Now after user registration instead of blindly saving everything to MongoDB we hashing our password. When it comes to checking correctness of password while logging in, we couldn't just compare passwords now. So we use **bcrypt.compare** function that hashes password entered by user and compare it with hashed password in database.

```
29  UserSchema.methods.isCorrectPassword = function(password, callback) {
30    // if (password === this.password) {
31    //   callback(null, true);
32    // } else {
33    //   callback(null, false);
34    // }
35    bcrypt.compare(password, this.password, function(err, same) {
36      if (err) {
37        callback(err);
38      } else {
39        callback(err, same);
40      }
41    });
42  }
43
44  module.exports = mongoose.model('User', UserSchema);      faizanv, 5 years ag
```

Commented code is what was before

So now after running script **sqli.js** we have:

```
[
  {
    _id: 643e799ee1e094bf943d65a7,
    email: 'check@mail.com',
    password: '$2b$10$eTaYGTx4XcymuiO9LtG/7.IQIvmuZk7OgBtkJDz9t6venlYooDzk.',
    isAdmin: true,
    __v: 0
  },
  {
    _id: 643e7a223df693c0b1178f14,
    email: 'bob@mail.com',
    password: '$2b$10$w.B90JgW6BcPjLuz9rqtHOmiWAAZP8YNGPUp/3aW8VbouDtCUDNo.',
    isAdmin: false,
    __v: 0
  },
  {
    _id: 643e7a413df693c0b1178f17,
    email: 'minecrafter34@mail.com',
    password: '$2b$10$lB1zMoPvXLq5u5k346zoE.N9QMUgAbtMCFFo7ZtNRHiR7OiU.pJO.',
    isAdmin: false,
    __v: 0
  }
]
```

```
94          } else {
95            const isAdmin = user.isAdmin
96            const payload = { email, isAdmin };
97            const token = jwt.sign(payload, secret, {
98              expiresIn: '1h'
99            });
100           res.cookie('token', token, { httpOnly: true }).sendStatus(200);
101         }
102       });
```

Let’s now encrypt model name using crypto lib:

```
41
42  const cipher = crypto.createCipher('aes256', secret);
43  const modelName = cipher.update(key, 'utf8', 'hex') + cipher.final('hex');
44  console.log('modelName', modelName)
45
46  const User = mongoose.model(modelName, UserSchema);
47        You, 2 seconds ago • Uncommitted changes
48  module.exports = User;

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  GITLENS

maruf@lenovo-maruf:~/university/ncs/react-auth-example(master)$ npm run server

> react-auth-example@1.0.0 server /home/maruf/university/ncs/react-auth-example
> node server.js

modelName b0947fe165bb43faef6d98e25ae48255
```

Now it would be very hard to brute force such model name and we can work with Database model only by importing it.

XSS

Description

XSS (Cross-Site Scripting) is a type of security vulnerability in web applications where an attacker injects malicious code into a web page viewed by other users. This can be done through input fields, such as search boxes or comment sections, or through URLs. When a user visits the page, the malicious code executes in their browser and can steal sensitive information, such as login credentials or personal data. XSS attacks can also be used to deface websites or spread malware. To prevent XSS attacks, web

developers should sanitize user input and encode output to prevent malicious code from being executed.

Vulnerability

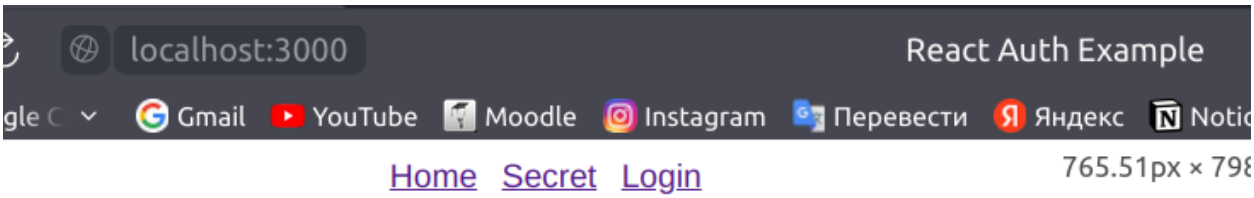
Even though React is mostly save from XSS attacks, there is scenario where such vulnerability could be exploited. Suppose in **api/home** get request we want to send html-markup with greeting:

```
35  app.get('/api/home', withAuth, function(req, res) {
36    const userStatus = req.isAdmin ? 'admininistrator' : 'user'
37    const body = `
38      <p>Welcome, dear ${userStatus} </p>
39      <p>With email <b>${req.email}</b> </p>
40    `
41    res.send({ body });
42  });      faizanv, 5 years ago • Initial commit …
```

And in the client side handle it this way:

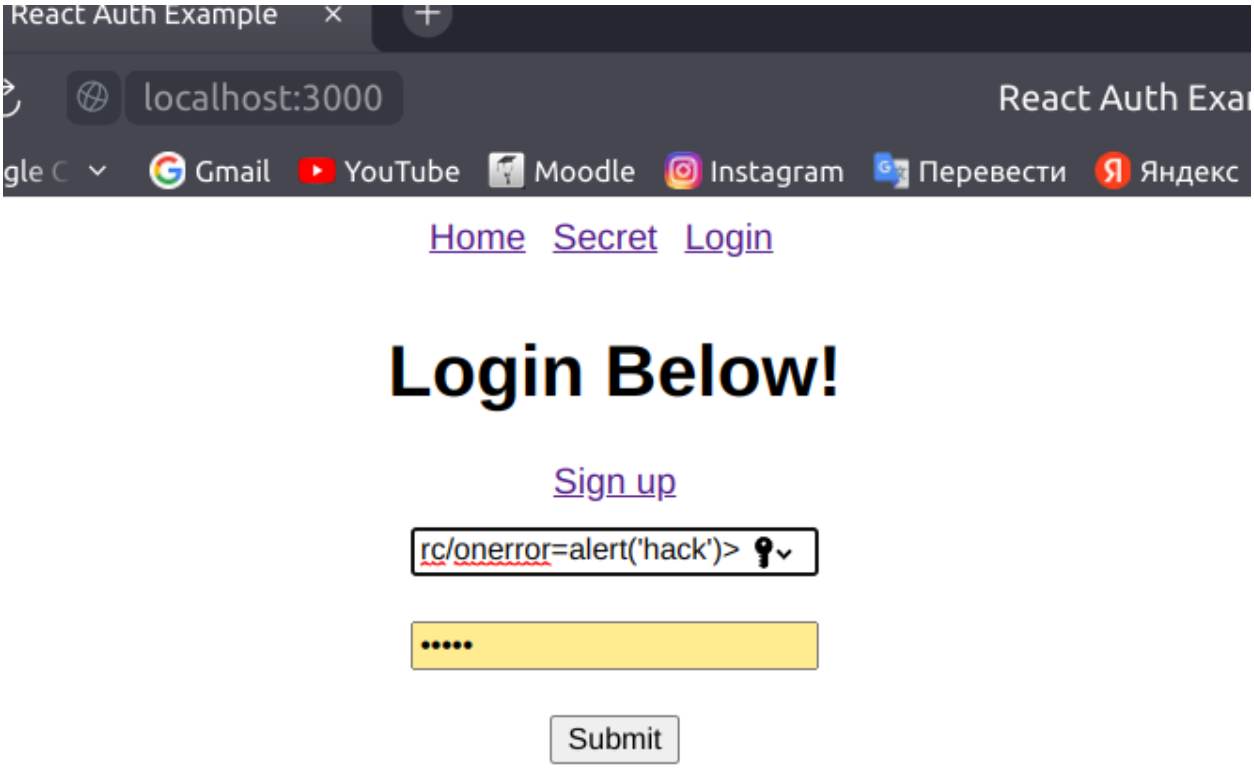
```
11  componentDidMount() {
12    fetch('/api/home')
13      .then((res) => res.json())
14      .then((res) => this.setState({ body: res.body }));
15  }
16
17  render() {
18    const HomeContent = `
19      <h1>Home</h1>
20      ${this.state.body}      You, 1 second ago • Uncommitted changes
21    `
22
23    return <div className="wrap" dangerouslySetInnerHTML={{__html:HomeContent}} />
24  }
25 }
26
```

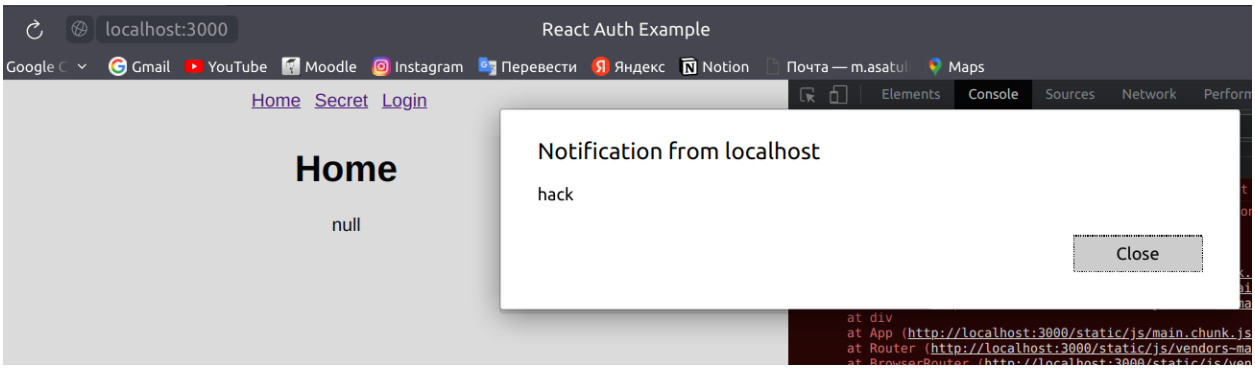
So we see this output:



However we should be careful with handling user email

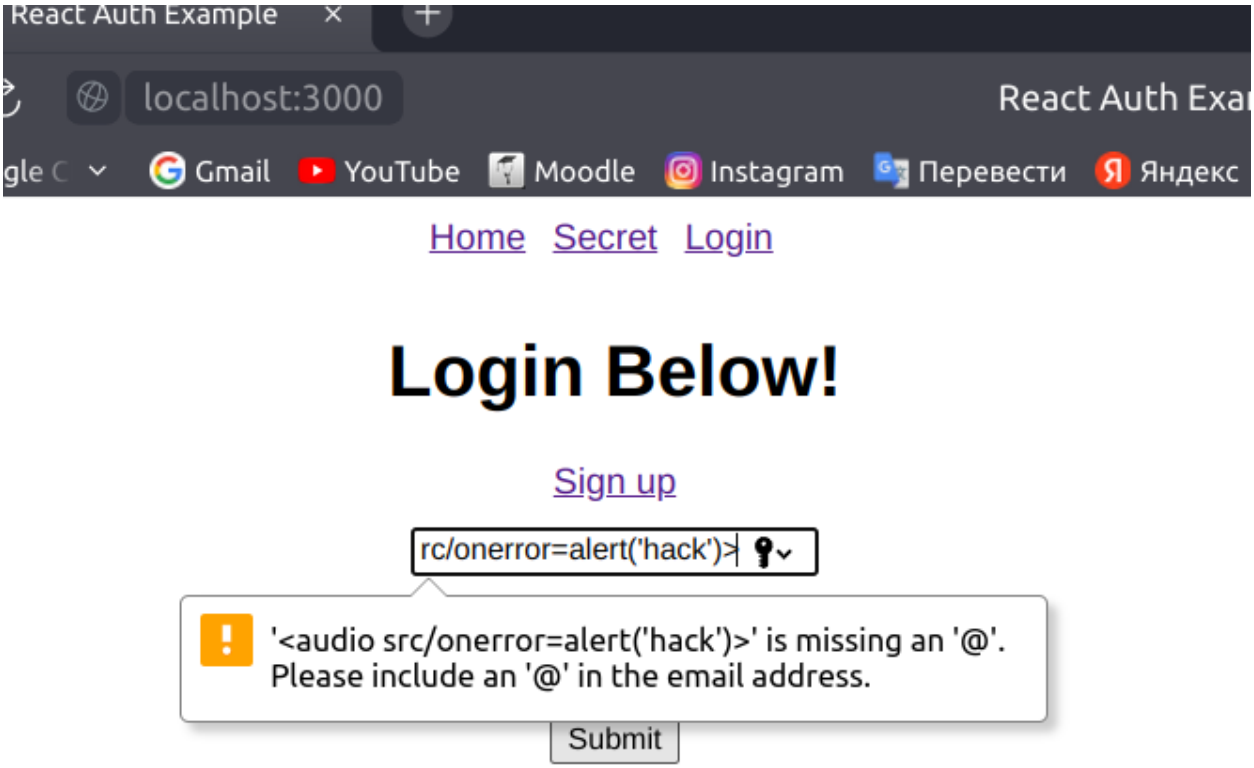
Exploitation





How to fix

First of all, validate input, restrict entering suspicious symbols like ‘<’ ‘>’.




```
72
73  const hasRestrictedSymb = (str) => {
74    const restrictedSymbs = ['<', '>', '"', "'", '/', '\ '.trim(), ';', ':', '&', '%', '+', '`'];
75    restrictedSymbs.forEach((symb) => {
76      if (str.indexOf(symb) > -1) {
77        return true
78      }
79    })
80    return sfalse
81  }      You, 2 seconds ago • Uncommitted changes
82
```

```
98      } else if (hasRestrictedSymb(email) || hasRestrictedSymb(password)) {
99        res.status(401)
00          .json({
01          error: 'There are some restricted symbols in email/password'
02        });
```

Then not pass html markup and don’t allow any markup job on the server side

```
34
35  app.get('/api/home', withAuth, function(req, res) {
36    const userStatus = req.isAdmin ? 'admininistrator' : 'user'
37
38    res.send({ status: userStatus, email: req.email });
39  });
```

```
12    componentDidMount() {
13      fetch('/api/home')
14        .then((res) => res.json())
15        .then((res) => this.setState({ status: res.status, email: res.email }));
16    }
17
18    render() {
19      const HomeContent = `
20        <h1>Home</h1>
21        <p>Welcome, dear ${this.state.status} </p>
22        <p>With email <b>${this.state.email}</b> </p>
23      `
```