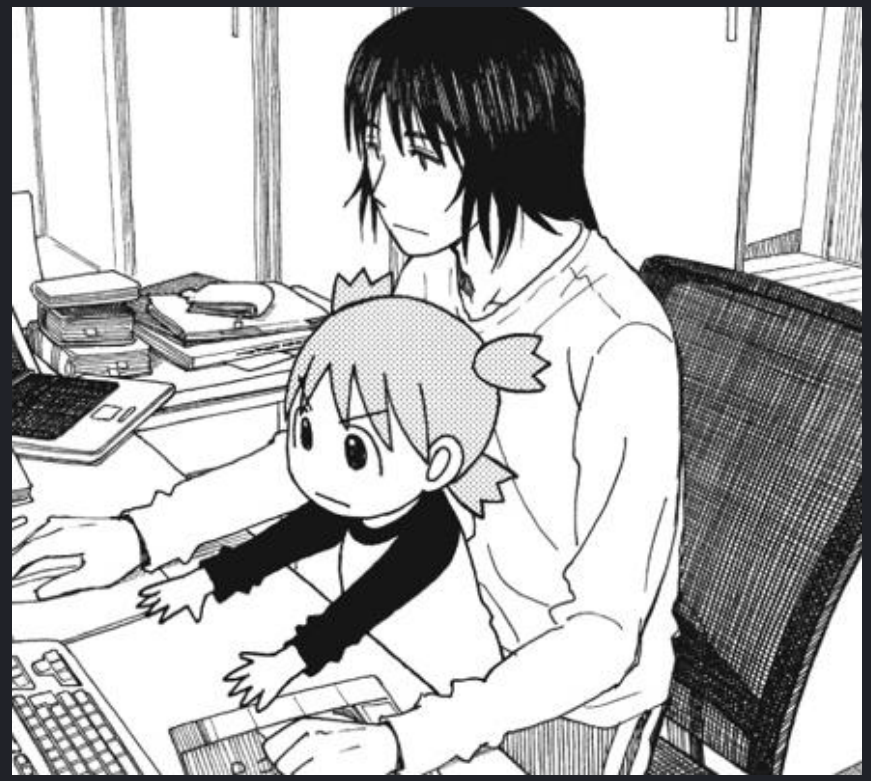
Breaking Access Control and Authentication in Web CTFs

Marcus Chan
@benkyou

$ whoami

- Marcus Chan
- Computer Science @ USM
- Started playing CTFs in 2024 solo
- CTFs @ RE:UN10N
  - *Mostly web but I play a bit of everything :D*



Why you should try web! 🕸️

- 🤗 Low barrier to entry
- 🤖 There's a lot of web technologies, like seriously **A LOT**
- 🌍 Most apps/services are web-based

Our goals for today

1. Understand how access controls and authentication can go wrong in applications
2. Identify these vulnerabilities and exploit them
3. Patch the vulnerabilities

What even is access control?

- Managing who is allowed to do certain actions or access certain resources
- Protect something sensitive, critical

Common Scenarios for Access Control Bugs

- Application has multiple roles, and the goal is to escalate privileges to perform some action
- You can only register as a low privileged user
- You need to access someone else's account

How to test for access control bugs?

1. Just browse to the *protected* page

```
https://vulnerable.com/admin <-- 403 Forbidden
https://vulnerable.com/admin/dashboard <-- 200 OK 🤩
```

- Sensitive functionality can be accessed by anyone who has the URL
- You should test this with unauthenticated and authenticated (low-priv) sessions

2. Change parameters in URL to access another object

```
https://vulnerable.com/user?id=1 <-- My info
https://vulnerable.com/user?id=1337 <-- Admin's info
```

- Discloses sensitive information of other users
- If the ID is unconventional, figure out how it's being generated
- We call this an **Insecure Direct Object Reference (IDOR)**

3. Understand how privileges are assigned in the application

- Maybe there's a bug in registration that allows you to register as an admin
- Find some other way to take over the admin's account

Our goal is to break in

Demo 1

Let's exploit an access control bug to get our first flag :D

What is Authentication?

- All the processes that verify a person's identity
- Are you really who you say you are?

A lot can go wrong in authentication

- Weak passwords. admin, admin123, password123
- Broken logic in the implementation 😉

Session Management

- The app needs to keep track of the user's state after logging in
- Different options:
  - Cookies, i.e: `JSESSIONID`, `PHPSESSID`, `ASP.NET_SessionId`, etc.
  - Authorization header
  - Token stored client side only

How do we test for broken authentication?

- Brute-force 🗿 THIS IS VERY UNLIKELY
- Go through the logon/signup process in unexpected order
- Password resets

```
https://vulnerable.com/reset-password?email=benkyou@cslu.com
```

- Token reuse

```
https://vulnerable.com/reset-password?email=benkyou@cslu.com&token=a0ba0d1c
```

Demo 1 Part 2

Exploit a broken auth logic in the app to become admin

So what went wrong?

Identifying the IDOR

app.py (line 100-111)

```
@app.route('/user/<int:id>')
def get_user(id):
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute('SELECT id, name, email, company, job_title, job_description, document_url, role FROM users WHERE id = %s', (id,))
    user = cursor.fetchone()
    cursor.close()
    conn.close()

    if user:
        return jsonify(user)
    return jsonify({'error': 'User not found'}), 404
```

- Fetch without validating who requested the user's info

Multiple ways to "fix"

1. If only the owner can access the user's info, then verify the user's session

```
@app.route('/user/<int:id>')
def get_user(id):
    if session.get('user_id') != id:
        return jsonify({'error': 'Unauthorized'}), 403
// existing code
```

2. If we want user info to be public, then remove the sensitive info.

```
@app.route('/user/<int:id>')
def get_user(id):
    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute('SELECT id, name, email, company, job_title, job_description, role FROM users WHERE id = %s', (id,))
    user = cursor.fetchone()
    cursor.close()
    conn.close()
```

Broken password reset logic

User controls which email to reset password

```
@app.route('/reset-password', methods=['GET', 'POST'])
def reset_password():
    if request.method == 'GET':
        if 'user_id' not in session:
            return redirect(url_for('login'))
        return render_template('reset_password.html', user={'email': session.get('user_name')})

    data = request.get_json()
    email, new_password = data.get('email'), data.get('password')
...[SNIP]...
    try:
        cursor.execute('UPDATE users SET password = %s WHERE email = %s', (hashed_password, email))
        conn.commit()
        return jsonify({'message': 'Password updated successfully'}), 200
...[SNIP]...
```

Fix: User should only reset their own account

```python
@app.route('/reset-password', methods=['GET', 'POST'])
def reset_password():
    if request.method == 'GET':
        if 'user_id' not in session:
            return redirect(url_for('login'))
        return render_template('reset_password.html', user={'email': session.get('user_name')})

    data = request.get_json()
    new_password = data.get('password')

    if not new_password:
        return jsonify({'error': 'Missing required fields'}), 400

    # Get the current user's id
    id = session.get('user_id')

    conn = get_db_connection()
    cursor = conn.cursor(dictionary=True)
    cursor.execute('SELECT * FROM users WHERE id = %s', (id,))
    user = cursor.fetchone()

    if not user:
        return jsonify({'error': 'User not found'}), 404

    hashed_password = bcrypt.hashpw(
        base64.b64encode(hashlib.sha256(new_password.encode()).digest()),
        bcrypt.gensalt()
    )

    try:
        cursor.execute('UPDATE users SET password = %s WHERE id = %s', (hashed_password, id))
        conn.commit()
        return jsonify({'message': 'Password updated successfully'}), 200
    except Exception as e:
        return jsonify({'error': str(e)}), 500
    finally:
        cursor.close()
        conn.close()
```

JSON Web Tokens (JWT)

- Token stores info about the user for authentication, session handling, access control
- Very common in API authentication

Anatomy of a JWT

- Consists of 3 parts – header, payload, signature
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImJlbmt5b3UiLCJyb2xlIjoiYWRtaW4iLCJpYXQiOjE1MTYyMzkwMjJ9.5q-uiE81uV6qWoxxcOtlhxt6Ir9i88yyyo-xWYEl53w

Each section is base64 encoded.

Header:

```
{
    "alg":"HS256",
    "typ":"JWT"
}
```

This specifies the type of algorithm used - none, symmetric (HS256), asymmetric (RS256)

Payload:

```
{
  "sub": "1234567890",
  "name": "benkyou",
  "role": "admin",
  "iat": 1516239022
}
```

Only server has the signing key, so it can verify the signature to prevent tampering

How do JWT vulnerabilities arise?

- Failing to verify the signature

jwt.decode(token [, options])

(Synchronous) Returns the decoded payload without verifying if the signature is valid.

> **Warning:** This will **not** verify whether the signature is valid. You should **not** use this for untrusted messages. You most likely want to use `jwt.verify` instead.

- `none` algorithm supported
  - eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImJlbmt5b3UiLCJyb2xlIjoiYWRtaW4iLCJpYXQiOjE1MTYyMzkwMjJ9.
- Weak signing key / Key is compromised
- Algorithm confusion attacks

My Methodology for JWT bugs

1. Start with the simplest cases.
    - Does the server even verify your token?
2. For HS256, can the key be brute-forced?
    - Can you steal the key from the server? i.e LFI
3. For RS256
    - Can you trick it to use HS256?
    - Can you inject your own JWT headers?

Demo 2

Let's exploit a broken JWT implementation

So what went wrong?

Weak HMAC Signing Key

middleware/token.js

```
const jwt = require('jsonwebtoken');

const SECRET_KEY = 'insert_your_random_key_here';
```

- Weak signing key is used so we can brute force
- Bad practice to hard-code keys in your code

The Fix

Generate a random secret key

```
require('crypto').randomBytes(32).toString('hex')
// a6365557e93b624ff591d5b83fc9223eb6020207009c2ce56f6897dd7799f90b
```

Set your key in environment variables and read it in your code

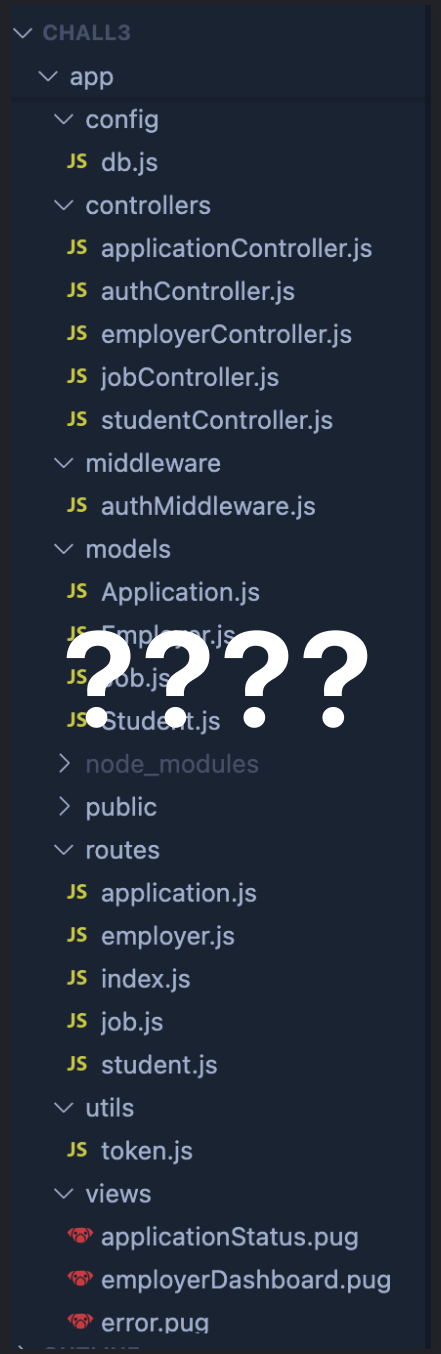
```
const SECRET_KEY = process.env.SECRET_KEY;
```

Now it's your turn!

Test your knowledge with challenge 3.

I heard benkyou's company is looking for a new intern and they're paying a lot. Internship application deadline is approaching but nobody has been accepted yet... or so it seems.

Will you be their new hire?


CHALL3
app
config
db.js
controllers
applicationController.js
authController.js
employerController.js
jobController.js
studentController.js
middleware
authMiddleware.js
models
Application.js
????
node_modules
public
routes
application.js
employer.js
index.js
job.js
student.js
utils
token.js
views
applicationStatus.pug
employerDashboard.pug
error.pug

MVC Pattern

- **M**odel: Manages data and business logic (your database operations)
- **V**iew: The user interface
- **C**ontroller: Middleman between Model and View

Broken password reset logic for Employer

controllers/authController.js (line 73-87)

```
const resetEmployerAccount = asyncHandler(async (req, res) => {
...[SNIP]...
    const { email, recoveryCode } = req.body;
    const success = await Employer.reset(email, recoveryCode);
...[SNIP]...
});
```

models/Employer.js (line 69-79)

```
static async reset(email, recoveryCode) {
    try {
        const [result] = await pool.query(
            'UPDATE employer SET is_activated = 1 WHERE email = ? AND recovery_code = ?',
            [email, recoveryCode]
        );
        return result.affectedRows > 0;
    } catch (error) {
        throw error;
    }
}
```

- Assumption that the account should be activated if it needs to be reset

controllers/authController.js (line 120-146)

```
const loginWithOTP = asyncHandler(async (req, res) => {
    const { encodedEmail, otp } = req.params;

    try {
        const email = Buffer.from(encodedEmail, 'base64').toString();
        const employer = await Employer.verifyOTP(otp, email);
```

- Base64 decoded email and OTP in URL parameters are passed to `Employer.verifyOTP(otp, email)`

Our IDOR bug is here

models/Employer.js (line 103-122)

```
static async verifyOTP(otp, email) {
    try {
        const [rows] = await pool.query(`
            SELECT e.* FROM employer e
            INNER JOIN employer_otp eo
            WHERE eo.otp_value = ?
            AND eo.expires_at > NOW()
            AND e.email = ?
            LIMIT 1`,
        [otp, email]);
...[SNIP]...
```

Looks good to me 🤔

Let's try running the query using our OTP with another employer

```
SELECT e.* FROM employer e
INNER JOIN employer_otp eo
WHERE eo.otp_value = "8c1b3bzb0va"
    AND eo.expires_at > NOW()
    AND e.email = "meow@meowcore.com" LIMIT 1;
// MeowCore Sdn Bhd returned 🤯
```

- The original query is missing the `ON` clause
- As a result, this doesn't enforce the relationship between an employer and their own OTP!

The Fix

With `ON`, the OTP is tied to the Employer through the employer's id

```
SELECT e.* FROM employer e
INNER JOIN employer_otp eo ON e.id = eo.employer_id
WHERE eo.otp_value = '8c1b3bzb0va'
    AND eo.expires_at > NOW()
    AND e.email = 'meow@meowcore.com';

Empty set (0.00 sec)
```

OTP can now only be used by its owner

```
SELECT e.* FROM employer e
INNER JOIN employer_otp eo ON e.id = eo.employer_id
WHERE eo.otp_value = '8c1b3bzb0va'
    AND eo.expires_at > NOW()
    AND e.email = 'benkyou@cslu.com';
// benkyou returned
1 row in set (0.00 sec)
```

Takeaways

- Bugs don't always have to be complicated
- Look out for weird behaviours
- Break the app's flow in unexpected ways

Any questions?

Thanks everyone!

Discord @benkyou

Twitter @benkyou_twt

https://www.linkedin.com/in/mcjhao

mail@mcjhao.xyz